SPARK Examiner

The SPARK Ravenscar Profile

Ravenscar
Issue: 1.6
Status: Definitive
25 January 2008

Originator

SPARK Team

Approver

SPARK Team Line Manager

Contents

1 Introduction 3

2 An overview of the SPARK Ravenscar profile 4
2.1 Introduction to the Ada Ravenscar profile 4
2.2 Potential errors in using the Ravenscar Profile 5
2.3 Introduction to RavenSPARK 7

3 RavenSPARK in detail 9
3.1 Supporting environment 9
3.2 Additional annotations 15
3.3 Language constructs 16
3.4 Flow analysis 35
3.5 Conversion of existing sequential SPARK programs 39
3.6 Proof support 39

4 Examples 41
4.1 Simple stopwatch 41
4.2 Mine pump 47

A Appendix: RavenSPARK templates and building blocks 48

B Syntax summary 62

C RavenSPARK design rationale 65

Document Control and References 72
Changes history 72
Changes forecast 72
Document references 72

1 Introduction

The Ravenscar Profile provides a subset of the tasking facilities of Ada 95 suitable for the construction of high-integrity concurrent programs. The Profile will become part of the Ada language standard with the Ada 0Y revision. For a full description of the Ravenscar Profile plus the rationale behind the choice of features, and several detailed examples, see the Ravenscar Guide technical report [1].

The Profile says nothing about the *sequential* parts of the code in the program; however, its adoption is not sufficient to eliminate certain forms of erroneous or implementation-defined behaviour associated with Ada 95 concurrency. Furthermore, the Profile defines a number of new errors which must be detected at run time.

SPARK is a well-established sequential subset of Ada. Combining the best aspects of the Ravenscar Profile (giving deterministic concurrency) with SPARK (to ensure predictable sequential execution) provides the means to construct highly-reliable concurrent programs. Extensions to the analyses performed by the SPARK Examiner provide a way of statically eliminating all forms of undesirable behaviour defined by the Profile as outlined in the previous paragraph.

The new, concurrent features of SPARK are entirely optional. If not selected, then the definition of the sequential SPARK language and the behaviour of the SPARK Examiner are unaffected. Where it is necessary to distinguish between sequential SPARK and the new language features, we use the term *RavenSPARK*™. The term SPARK encompasses both the existing sequential SPARK language and RavenSPARK. Throughout this document Ravenscar (with an initial upper-case letter) means the profile itself as defined in [1]; ravenscar (with a small r) means the Examiner's analysis profile of that name as selected by the appropriate command line switch (see 3.1.1).

This document is in three main sections. The first provides an overview of the Ravenscar profile and identifies some problematic issues affecting its use. The second section provides a detailed description of RavenSPARK. The last section and an accompanying appendix provides sample RavenSPARK code including templates for commonly occurring programming constructs. Readers familiar with tasking in Ada, especially those familiar with the Ravenscar profile, may find it useful to look at the examples and templates first before checking the detailed rules in sections 3.3 to 3.6. Readers are assumed to have some familiarity with sequential SPARK and the SPARK Examiner toolset.

2 An overview of the SPARK Ravenscar profile

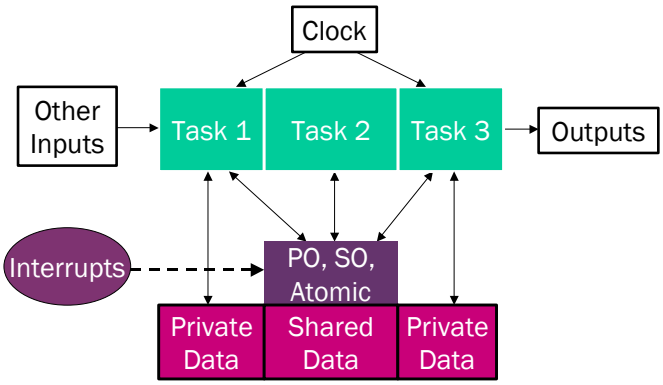
2.1 Introduction to the Ada Ravenscar profile

2.1.1 Ravenscar components

The Ravenscar Profile is fully defined in section 3.2 of [1]. The Profile is enabled in the compiler by means of a set of pragma Restrictions and other configuration pragmas. The language subset defined by these pragmas allows construction of concurrent programs comprising:

- a fixed set of non-terminating, library-level *tasks*;
- a fixed set of shareable global variables which are either:
  - *protected objects*; or
  - objects of a *pragma Atomic* type;
- a fixed set of global objects allowing synchronization of tasks:
  - protected objects with a single *entry*; or
  - predefined *suspension objects*;
- high-precision timing facilities; and
- deterministic scheduling and locking policies.

2.1.2 Real-time program model



The diagram shows the interaction of the components of a sample Ravenscar program. Task communication is achieved by the use of protected, shared data which may take the form of a protected object, suspension object or an object of a pragma Atomic type. Tasks may access unprotected data provided they do not do so concurrently with other tasks (see 2.2.2.1 below). These rules ensure there is no contention or concurrent access to data leading to incomplete reads or writes and consequent data corruption. Tasks may also make use of the high-precision clock to run at set periodic rates and/or may run aperiodically depending on the state of the protected objects they use.

Interrupts may arrive and influence the execution of tasks via handlers implemented as protected subprograms. Tasks and protected objects may have *priorities* determining their importance and influencing when they will be executed.

Protected objects are locked during access preventing concurrent access; mutual exclusion is guaranteed by a locking policy called *ceiling locking*. Within their priorities, tasks are dispatched on a first-come-first-served basis known as *FIFO within priorities* dispatching; this, together with a requirement that delays are based on absolute time rather than relative time, allows deterministic execution amenable to *schedulability analysis* (for example: *rate monotonic analysis* [3]).

2.2 Potential errors in using the Ravenscar Profile

Although it represents an enormous step forward in its support for reliable concurrent programming, the Profile identifies a number of error conditions which may give rise to run-time exceptions, erroneous behaviour or implementation-defined behaviour. Section 6 of [1] provides a detailed explanation of these problem areas and outlines theoretical approaches that could be taken to detect and eliminate them statically. We briefly restate the problems here before describing how RavenSPARK eliminates them.

2.2.1 Errors Leading to Run-time Exceptions

2.2.1.1 Effect of Unexpected Exceptions

The Ravenscar Profile does not address the occurrence of unhandled exceptions since exceptions relate to the sequential, rather than the concurrent, part of the language. Nevertheless, whereas an unhandled exception will cause a sequential program to terminate, an unhandled exception during the execution phase of a concurrent program is not so readily detected. In particular, an unhandled exception can cause any of the following effects:

- silent abandonment of the execution of an interrupt handler;
- silent termination of a task;
- premature exit from a protected action, possibly leaving it in an inconsistent state.

The SPARK Examiner already provides the necessary facilities to show proof of absence of run-time errors in sequential code and these techniques are strongly recommended to be applied to the sequential code within each task and protected object, leaving only the possibility of the exceptions that relate directly to the concurrency behaviour.

2.2.1.2 Exceptions Due to Concurrency

Of the concurrency checks defined by Ada95, there are only two that apply to a Ravenscar Profile program:

1 detection of priority ceiling violation as defined by the Ceiling_Locking policy which requires that calls from a task to protected operations must follow a non-decreasing priority chain;

2 detection of violation of not more than one task waiting concurrently on a suspension object (via the Suspend_Until_True operation).

In addition, two further concurrency checks are introduced by the Ravenscar Profile definition:

1 the maximum number of calls that are queued concurrently on a protected entry shall not exceed one;

2 a potentially blocking operation shall not be executed by a protected action.

2.2.2 Errors Leading to Erroneous Behaviour

2.2.2.1 Use of Unprotected Shared Variables

If two tasks share an unprotected variable then the execution of the resulting program may be erroneous. In principle this does not prevent the sharing of such variables; however this would require a

demonstration that the temporal properties of the program prevented concurrent access to it. More robust protection requires a check that unprotected data can never be shared.

2.2.2.2 Race Conditions During Elaboration

Sequential SPARK eliminates all dependency on elaboration ordering. Section 6.2.2 of [1] provides a detailed explanation of why the situation is less clear cut in the case of concurrent programs and, in particular why there is a risk of race conditions during elaboration. Essentially the problem is that library-level tasks are activated in package-elaboration order, and execute before all library-level data has been elaborated. Thus a task may access global data before the elaboration code has initialized it.

These problems can be eliminated by the use of a new Partition_Elaboration_Policy pragma which has been agreed for addition to the next revision of the Ada language standard. This pragma has the effect of holding back task execution until all elaboration code had finished running; however, this pragma is *not* part of the Ravenscar Profile and not yet part of the Ada language.

2.2.2.3 Program Incompleteness

It is not easy to ensure that all the tasks and protected objects required to provide a program's intended behaviour have actually been included in its executable image. Unlike sequential code, the failure to "with" an active component does not make the program illegal; it simply becomes a legal but different program performing a subset of its intended action.

2.2.3 Errors Leading to Implementation-defined Behaviour

2.2.3.1 Task Termination

The Ravenscar Profile includes the Restrictions pragma No_Task_Termination, but the dynamic effect of such termination is implementation-defined. Attempts at adding a formal task termination handling mechanism to the next revision of the Ada language standard are at an immature state of development. A means of showing that tasks will *not* terminate is therefore essential if implementation-defined behaviour is to be avoided.

2.3 Introduction to RavenSPARK

RavenSPARK is constructed in exactly the same way as the sequential SPARK language and with exactly the same objectives. A combination of language restrictions (RavenSPARK subsets the Ravenscar subset!) and additional annotations allows the construction of programs which comply with the Ravenscar Profile but which also have the property that it is possible to show, statically, without requiring a complete program closure, that all of the errors described in section 2.2 have been eliminated. In addition, the analysis facilities of sequential SPARK are retained and additional, partition-wide forms of analysis introduced.

RavenSPARK is supported by a new analysis profile of the SPARK Examiner which is enabled with a command-line switch.

3 RavenSPARK in detail

This section provides an informal description of RavenSPARK. The definitive rules of this variant of the SPARK language are included in [2] which is included in the SPARK Examiner documentation set. Appendix C contains rationales for the various restrictions imposed by RavenSPARK; these are referenced in the main text by a superscript "R" and an identifying number.

3.1 Supporting environment

3.1.1 Command line

RavenSPARK follows Ada's lead by regarding the concurrent program features as a *profile*. The command line options have been extended to allow selection of an *analysis profile*. There are currently two supported profiles: *sequential* and *ravenscar*. Sequential is the default and gives access to the sequential SPARK language familiar to SPARK users. Selecting the ravenscar analysis profile gives access to the new, concurrent language features.

The new command line switch is *profile* and the syntax is as follows:

Option	Syntax	Default	Description
profile	/pr= *type* (-pr on Unix)	sequential	Selects a particular "profile" of the SPARK language which will be used during analysis. *type* may be *sequential* or *ravenscar* and may be abbreviated to *s* or *r* respectively. The form of the annotation allows for the possibility of adding other analysis profiles to the SPARK Examiner in the future.

3.1.2 New reserved words

There are no new reserved words when the Ravenscar profile is enabled since all reserved words of Ada (including those associated with tasking) are automatically reserved words of SPARK. However, certain reserved words are accepted by the Examiner when processing RavenSPARK which are rejected[1] when processing sequential SPARK. The recognised reserved words are as follows:

Reserved word	Description
`task`	Used to introduce a task type (see 3.3.2) declaration or body and in extended form of the own variable annotation
`protected`	Used to introduce a protected type (see 3.3.3) declaration or body and in extended form of the own variable annotation

[1] Prior to Release 7 of the Examiner this rejection would take the form of a syntax error; after Release 7, a semantic error is issued.

Reserved word	Description
`entry`	Used to define a protected entry in a protected object on which a task may suspend until some event causes an entry barrier to be lifted
`when`	Used in a protected entry body to define the entry barrier
`delay until`	Used (always together) in a task body or procedure to indicate that the task should suspend to the absolute time specified in the statement (see 3.3.1)
`declare`	Used to provide additional information in the annotations of tasks and unprotected procedures. Declare has been used here in the sense of "I declare" meaning "I am admitting that"; it does not imply a declaration in the Ada sense and the reserved word is being overloaded in this context in preference to adding a new one to the SPARK language
`aliased`	The grammar of variable declarations has been extended to allow the use of the reserved word `aliased`. This is in support of planned work involving access discriminants and is currently only useful in conjunction with hidden code that uses the `'Access` attribute.

3.1.3 Predefined packages

Certain predefined child packages of package Ada are essential for the construction of Ravenscar programs. When the Ravenscar profile is enabled, the Examiner makes these packages available for use. The additional packages are predefined within the SPARK Examiner and are not therefore provided as source code and do not need to be lexically processed by the Examiner. The predefined packages enabled by the analysis profile are as follows.

3.1.3.1 Ada.Real_Time

This package provides access to the high-precision, real-time clock required by the Ravenscar profile and other operations to support timing activities. The specification of the SPARK version of the package is given below. Note that an own variable of mode `in` is declared to represent the stream of clock times arriving from the environment and that this own variable is also marked as being *protected* so that the clock can be shared.

```
package Ada.Real_Time
--# own protected in ClockTime : Time;
is
  type Time is private;
  Time_First : constant Time;
  Time_Last : constant Time;

  type Time_Span is private;
  Time_Span_First : constant Time_Span;
  Time_Span_Last : constant Time_Span;
  Time_Span_Zero : constant Time_Span;
  Time_Span_Unit : constant Time_Span;

  Tick : constant Time_Span;
  function Clock return Time;
  --# global ClockTime;
```

```
   -- the following operators are predefined
   -- Time + Time_Span return Time
   -- Time_Span + Time return Time
   -- Time - Time_Span return Time
   -- Time - Time return Time_Span

   -- Time <  Time return Boolean
   -- Time <= Time return Boolean
   -- Time >  Time return Boolean
   -- Time >= Time return Boolean

   -- Time_Span + Time_Span return Time_Span
   -- Time_Span - Time_Span return Time_Span
   --  - Time_Span return Time_Span
   -- Time_Span * Integer return Time_Span
   -- Integer * Time_Span return Time_Span
   -- Time_Span / Time_Span return Integer
   -- Time_Span / Integer return Time_Span

   -- abs Time_Span return Time_Span

   -- Time_Span <  Time_Span return Boolean
   -- Time_Span <= Time_Span return Boolean
   -- Time_Span >  Time_Span return Boolean
   -- Time_Span >= Time_Span return Boolean

   function Nanoseconds  (NS : Integer) return Time_Span;
   function Microseconds (US : Integer) return Time_Span;
   function Milliseconds (MS : Integer) return Time_Span;

   type Seconds_Count is range implementation-defined;
   -- a range may be supplied using the configuration file, see 3.1.4

   procedure Split(T  : in     Time;
                   SC :    out Seconds_Count;
                   TS :    out Time_Span);
   --# derives SC, TS from T;

   function Time_Of(SC : Seconds_Count;
                    TS : Time_Span) return Time;

private
   --# hide Ada.Real_Time
   -- we do not need to know the implementation details of this package for our analysis purposes
end Ada.Real_Time;
```

Note that the conversion functions involving type `Duration` (`To_Duration` and `To_Time_Span`) are not included since there is no support for relative delays in the Profile[R1]. Also, the constant `Time_Unit` is not supported since it is an implementation-defined named real number.

SPARK does not allow user-defined function calls to be made in elaboration code or to initialize objects and, prior to the definition of RavenSPARK there were no predefined functions that could be called. Package `Ada.Real_Time` now introduces predefined functions that may be used during elaboration as follows:

1 The function `Clock` may be used to initialize a *constant* declared in a library-level package[R2]. This concession facilitates the co-ordinated starting of tasks via an "Epoch" package — see A.4.

2 The functions `Nanoseconds`, `Microseconds` and `Milliseconds` are regarded as static and may be used in elaboration code and to initialize objects as long as the argument supplied to them is static[R3]. This facilitates the definition of task periods, see for example, section 4.1.

3.1.3.2 Ada.Synchronous_Task_Control

Package `Ada.Synchronous_Task_Control` declares a predefined type and operations that provide a convenient way of co-ordinating a set of tasks by stopping and starting them in response to events.

```
package Ada.Synchronous_Task_Control
is
  type Suspension_Object is limited private;

  procedure Set_True(S : in out Suspension_Object);
  --# derives S from ;
  --  Note the apparent mismatch between the parameter mode and the derives annotation[R4].
  --  This arises because the Ada run-time system needs to read S0 in order to determine
  --  whether any tasks should now be started whereas, for flow analysis purposes, we only
  --  need to record the fact that S0 is given a new value that does not depend on any import.

  procedure Set_False(S : in out Suspension_Object);
  --# derives S from ;
  --  See comment on operation Set_True

  procedure Suspend_Until_True(S : in out Suspension_Object);
  --# derives S from ;
  --  See comment on operation Set_True

private
  --# hide Ada.Synchronous_Task_Control;
  --  we do not need to know the implementation details of this package for our analysis purposes
end Ada.Synchronous_Task_Control;
```

Note that function `Current_State` is not supported[R5] since the function result is inherently unsafe i.e. the state of the suspension object may change prior to execution of the code that depends on the result of the call to `Current_State`.

Since all the predefined operations on suspension objects only export their parameter, it follows that suspension objects are only ever exports in thread-level annotations (see 3.4.1). Suspension objects may only be imports at the partition level (see 3.4.2).

It is worth noting that the operation `Set_False` has to be used with considerable care since it is extremely easy to generate race conditions with it. For example, if a task is given freedom to run via a `Set_True` operation and that permission is revoked via a `Set_False` call, then whether the task is successfully prevented from running depends entirely on subtle timing effects. We do not recommend the use of `Set_False` unless it is supported by suitable analysis such as model checking.

3.1.3.3 Ada.Interrupts

Package `Ada.Interrupts` provides a number of operations to support the dynamic allocation of interrupts. The Profile allows only the static attachment of interrupts and SPARK therefore provides a subset of the package required to support pragma Attach_Handler[R1] (see 3.1.5).

```
package Ada.Interrupts is
   type Interrupt_ID is implementation-defined discrete type;
   -- values may be supplied using the configuration file, see 3.1.4

private
  --# hide Ada.Interrupts;
  -- we do not need to know the implementation details of this package for our analysis purposes
end Ada.Interrupts;
```

3.1.4 Extensions to the configuration file

The configuration file is described in the Examiner User Manual. It affords a means of providing compiler-dependent data to the Examiner which can then perform more precise analysis. The capability of the configuration file system is extended so that information relevant to concurrent programs can be provided. Note that these items are ignored by the Examiner if the ravenscar analysis profile is not selected.

3.1.4.1 Package System

Values of permitted priority ranges may be supplied as in the following example:

```
subtype Any_Priority       is Integer      range  0 .. 31;
subtype Priority           is Any_Priority range  0 .. 30;
subtype Interrupt_Priority is Any_Priority range 31 .. 31;
```

A constant `Default_Priority` with the value `(Priority'First + Priority'Last)/2` is implicitly declared when `Any_Priority` is defined. If `Any_Priority` is defined then the subtypes `Priority` and `Interrupt_Priority` must also be defined.

If provided, the Examiner will use these values to check the validity of Priority and Interrupt_Priority pragmas as well as the priority property in annotations.

Priority values may be supplied whether or not the Ravenscar analysis profile is selected; this is to allow a single package System configuration to be used for both sequential and concurrent code.

3.1.4.2 Package Ada.Real_Time

The range of values for the implementation-defined type Seconds_Count may be provided, when the Ravenscar profile is selected, as in the following example.

```
package Ada.Real_Time is
   type Seconds_Count is range -2 ** 63 .. 2 ** 63 - 1;
end Ada.Real_Time;
```

If provided, the Examiner will use these values in the generation of range checks.

3.1.4.3 Package Ada.Interrupts

If the implementation-defined discrete type `Interrupt_ID` takes the form of an integer type then lower and upper bounds may be provided in the configuration file, when the Ravenscar profile is selected, as in the following example.

```
package Ada.Interrupts is
   type Interrupt_ID is range 1 .. 32;
end Ada.Interrupts;
```

The Examiner will use any supplied values in range checks; however, it does not currently check whether the second argument to pragma Attach_Handler is in the range of `Interrupt_ID`, see 3.1.5.3 and 3.3.3.2.

3.1.5 Additional pragmas

Four additional pragmas gain semantic significance and are recognized by the Examiner when the Ravenscar analysis profile is selected.

3.1.5.1 pragma Priority

Pragma `Priority` is used to set the priority of task types and the Environment task, and to set the ceiling priority of protected types. The pragma takes a single argument of type Integer for which a static value[R25] must be supplied. If ranges have been set, using the configuration file utility (see 3.1.4.1), for the priority subtypes in package `System` then the Examiner will check that the supplied argument is a legal member of type `System.Priority`. SPARK imposes restrictions on the placement of pragma `Priority` which are described in 3.3.2 and 3.3.3.

3.1.5.2 pragma Interrupt_Priority

Pragma `Interrupt_Priority` performs a similar function to pragma `Priority` but accepts values in the wider range of `System.Any_Priority` thus giving access to values which are higher than `System.Priority'Last`. Pragma `Interrupt_Priority`, with an argument in the range of `System.Interrupt_Priority`, is required for protected types that include the interrupt property and which make use of pragma `Attach_Handler`. See 3.3.3.2.

3.1.5.3 pragma Attach_Handler

Pragma `Attach_Handler` is used to mark a parameterless protected procedure as an interrupt handler to connect it to the underlying operating system's interrupt mechanism. The pragma takes two arguments: the name of the subprogram and an interrupt identifier of type `Ada.Interrupts.Interrupt_ID`. SPARK requires the pragma to immediately follow the subprogram to which it relates[R6]. The Examiner checks that the second argument is provided but, since it is of an implementation-defined discrete type, does not process it further. Clearly more than one interrupt handler should not be attached to a single interrupt identifier. As a reminder of this, the Examiner issues a warning at the point of declaration of each protected object that contains an interrupt handler. Examples of the use of pragma `Attach_Handler` can be found in 3.3.3.2.

3.1.5.4 pragma Atomic

Pragma `Atomic` may be applied to a local, scalar[R31] type or to a non-tagged record type with a single field that is a predefined type[R37] where this is accepted by the Ada compiler to be used. Objects of a pragma `Atomic` type can be safely shared between tasks therefore an own variable declared with the protected modifier may be implemented as a variable of a pragma `Atomic` type (see 3.3.3.7). Pragma `Atomic` may *not* be applied to an *object*[R7].

3.1.6 Additional attributes

There are no additional attributes available when the Ravenscar analysis profile is selected. Note that the attribute `E'Count` which gives the number of tasks presently queued on an entry, is not currently supported in RavenSPARK[R8].

In addition, the entry attribute `E'Caller` and the task attributes `T'Callable, T'Terminated, T'Identity` and `T'Storage_Size` are not supported[R1].

3.2 Additional annotations

RavenSPARK defines some additional annotations that are used by the Examiner in the detection of the errors described in section 2.2. The new annotations are explained where they are used in later sections on language constructs.

The annotations fall into the following 3 groups:

1 A qualifier that can be used to indicate that an own variable is a *task* or *protected* object.

2 A *property list* that may follow an own variable declaration to indicate certain properties of it.

3 A *declare* annotation that follows the *global* and *derives* annotation of a task or subprogram and can be used to introduce a *property list* for that task or subprogram.

For reference purposes, the additional annotations can be summarised as follows.

Property	Location	Argument	Meaning
priority	own protected	expression	announces the priority that the object will have when it is declared
suspendable	own protected	none	indicates that the object is a predefined suspension object or a protected object with an entry
interrupt	own protected	optional (see examples)	indicates that the object contains one or more interrupt handlers and gives (optionally) a user-defined name for them
protects	own protected	identifier list	shows that the unprotected variables in the identifier list are used solely by the protected object and can be treated exactly like protected elements
suspends	unprotected procedure or task type	identifier list	indicates that the operation suspends on the identifiers given (by calling an entry in a named protected object, for example)
delay	unprotected procedure	none	marks a procedure as potentially executing a delay statement (or other blocking action)

3.3 Language constructs

In this section we will introduce the new language constructs made available under the Ravenscar profile and show how they are typically used.

3.3.1 Delay statement

The delay statement has the following syntax:

```
delay_statement ::= delay until expression;
```

The delay statement pauses execution of the current task or procedure to an absolute time given by the expression which must be of type `Ada.Real_Time.Time`. A delay statement may appear in the sequence of statements of an *unprotected proce*dure or *task body*. Note this prohibits use in *functions*[R9] and *protected* operations[R1] (where it would cause illegal blocking of the current task).

The delay statement affects only the temporal properties of a program and therefore does not affect data and information flow. The flow relation for the statement is equivalent to:

derives null from *values referenced by expression*

Therefore a delay statement that references only values derived from constants is semantically identical, in flow analysis terms, to a null statement. If the expression references variables (including

the own variable `Ada.Real_Time.ClockTime` via function `Ada.Real_Time.Clock`) then those variables must be defined in order to avoid data flow errors; however, they are not otherwise used.

If the delay statement appears in a procedure (as opposed to a task body) then the procedure must declare the fact that it delays in a *declare* annotation[R10]. For example, the following is a valid declaration for a procedure which simply delays until a time passed in to it as a parameter.

```
procedure Wait_Until (T : in Ada.Real_Time.Time);
--# derives null from T;
--# declare Delay;
```

The body of the procedure might be:

```
procedure Wait_Until (T : in Ada.Real_Time.Time)
is
begin
   delay until T;
end Wait_Until;
```

The delay annotation is transitive [R10] and must be propagated up the call tree.

3.3.2 Tasks

A task can be viewed as being an autonomous "main program" that runs in parallel with the other tasks making up the complete program. A task is an *object* of a named task *type*[R17]. The Ravenscar Profile requires that task objects be declared directly in a library level package; in SPARK terms a task object is therefore a form of *own variable*. A task is not called or explicitly started, it runs continuously after the associated task object is activated.

3.3.2.1 Task type declaration

A task type declaration provides a template for task objects including a description of the information flow of task objects of that task type. SPARK requires task *types* to be declared in package specifications (either the visible or private part) to facilitate certain program-wide analyses[R11].

Syntax

The syntax of a task type declaration is as follows.

task_type_declaration ::=
task type *defining*_identifier [known_discriminant_part] task_type_annotation task_definition

task_type_annotation ::=
moded_global_definition [dependency_relation] [declare_annotation]

task_definition ::=
is priority_pragma {*apragma*} **end** *defining*_identifier **;**

priority_pragma ::=
pragma Priority **(**expression**);** |
pragma Interrupt_Priority **(**expression**);**

known_discriminant_part ::=
(discriminant_specification {**;** discriminant_specification})

discriminant_specification ::=
identifier_list **:** type_mark

declare_annotation ::=
--# declare property_list **;**

property_list ::= property {**,** property}

property ::=
name_property
| name_value_property

name_property ::=
delay
| identifier

name_value_property ::=
identifier **=>** aggregate | expression

Examples

Some examples follow. Firstly a simple periodic task that processes the internal state of package P in some way.

```
task type Timer
--# global in out P.State;
--# derives P.State from P.State;
is
   pragma Priority (10);
end Timer;
```

The priority could be set using a *discriminant* (which must, in SPARK, be static, discrete and without a default expression).

```
task type Timer2 (Pr : System.Priority)
--# global in out P.State;
--# derives P.State from P.State;
is
   pragma Priority (Pr);
end Timer2;
```

For an aperiodic task, we must declare the object that the task suspends on[R12] (appropriate objects for suspension are discussed in sections 3.3.3 and 3.3.3.6). The suspends property annotation allows the Examiner to eliminate the possibility of more than one task being queued on the same entry (see 2.2.1.2).

```
task type ProcessWhenReady (Pr : System.Priority)
--# global in out P.State; out DataReady;
--# derives P.State   from * &
--#         DataReady from ;
--# declare Suspends => DataReady;
is
   pragma Priority (Pr);
end ProcessWhenReady;
```

Static semantics rules

1 A priority pragma is required[R13]; it must have an argument [R13] and this must resolve to a static expression not containing any *operators* [R14], either directly or via a discriminant value.

2 A global annotation is mandatory[R15] (otherwise the task would not be able to do anything except consume clock ticks).

3 A derives annotation is required if information flow analysis is to be performed.

4 A task (other than the environment task) may not import unprotected state variables; furthermore, the unprotected variables it exports may not appear in the initializes clause of the package in which they are declared [R16]. Essentially this means that a task is solely responsible for initializing any unprotected state that it uses; this is sufficient to avoid race conditions during elaboration.

5 The declare annotation may declare only the Suspends property; this is mandatory if the task suspends [R12] by either calling a protected entry or by calling `Ada.Synchronous_Task_Control.Suspend_Until_True` on an object of the predefined suspension object type. Note that if the task may suspend on more than one such object then the annotation takes the form of an aggregate: `declare Suspends => (SO1, SO2, PO1);`

3.3.2.2 Task subtypes

Where a task type declaration has a known discriminant part, a named subtype[R17] that provides the actual discriminant values must be declared before any task objects may be declared. A subtype of type Timer2 that is equivalent to the type Timer would be:

```
subtype SameAsTimer is Timer2 (10); -- priority set to 10 via the discriminant
```

A task *subtype* may be declared in a library package *specification* or *body*. Only task *types* are required to be declared in library package *specifications*[R11]. Actual discriminants must be *static* values[R25].

3.3.2.3 Task object declaration

A task object declaration is simply an object declaration. For example:

```
TheTimer : SameAsTimer;
```

Task objects must be declared directly in library level packages so task types cannot be used as part of a composite type, nor a formal parameter type, nor function return type[R35].

Because a task object is an *own variable* it must appear in its package's own variable annotation. The syntax of the own variable annotation has been extended to indicate which own variables are tasks (or protected objects, see 3.3.3) and to provide properties of the object.

own_variable_clause ::=
--# own own_variable_specification {own_variable_specification}

own_variable_specification ::=
own_variable_list [**::=** type_mark] [**(**property_list**)] ;**

own_variable_list ::=
own_variable_modifier own_variable { **,** own_variable_modifier own_variable }
own_variable_modifier ::= mode | **task** | **protected** | **protected in** | **protected out**

The key thing to remember in the use of the modifiers (e.g. *protected* or *out*) is that they only affect the immediately adjacent identifier. So

```
--# own in X, Y : Integer;
```

declares two own variables and announces them to be of type `Integer` but only X is of mode `in`.

For a *task object* the only requirement is that the own variable is type announced with its task *type* (not subtype)[R18]. Our declaration of `TheTimer` above must therefore be supported by an own variable annotation of the form:

```
--# own task TheTimer : Timer2;
```

An *own task* variable may not appear in a package body's *refinement* annotation.

3.3.2.4 Task bodies

A task body provides the sequence of statements that perform the actions of objects of the task type; these actions must match the annotation provided for the task type. One body is associated with each task type (not subtype or object). Task bodies may be placed in package bodies or may be subunits in which case a suitable stub appears in the package body.

Syntax

The grammar of a task body is as follows.

task_body ::=
task body *definining*_identifier procedure_annotation **is**
subprogram_implementation **;**

and for a task body stub:

task_body_stub ::=
task body *definining*_identifier procedure_annotation **is separate ;**

Static semantic rules

1 As for subprograms, a second annotation is required if the annotation on the task type refers to variables which are refined in the package body where the task body is declared[R19]. This second annotation will comprise a global and, possibly, a derives annotation but neither proof contexts nor a declare annotation.

2 The last statement of a task body must be a plain loop with no exit statements[R20].

Example

A possible body for task type Timer might be:

```
task body Timer
-- no second annotation required, no refinement elements used
is
   NextPeriod : Ada.Real_Time.Time;
begin
   NextPeriod := Epoch.TimerStart; -- package Epoch defines timing constants
   loop
      delay until NextPeriod;
      P.ProcessTheState;
      NextPeriod := NextPeriod + Epoch.TimerPeriodicity;
   end loop;
end Timer;
```

Further examples of typical idiomatic forms of tasks can be found in appendix A.

3.3.3 Protected objects

In RavenSPARK we allow package own variables to be marked as being *protected*. A protected own variable may be implemented in one of the following ways:

1 As a *protected object* of a named *protected type* (or subtype). Such a protected object provides an encapsulation of data and operations which ensures that mutual exclusion is enforced whenever the data is accessed. Protected objects have some analogies with packages and some with variables. The operations of a protected object are known as *protected operations* and may comprise *functions*, *procedures* and *entries*. User-defined protected types are described in sections 3.3.3.1 to 3.3.3.5 below.

2 As an object of the predefined type Ada.Synchronous_Task_Control.Suspension_Object. This is a useful predefined type which is provided for simple task co-ordination purposes; suspension objects are described in section 3.3.3.6.

3 As a variable of an atomic type. Such data is not protected by means of any locking mechanism but accesses to it are intrinsically safe because they are atomic at the machine code level. This form of protected state is described in section 3.3.3.7.

3.3.3.1 Protected types

Protected type declarations must be placed in library-level package specifications (visible or private part) in the same way as task type declarations[R21].

Syntax

The syntax of a protected type declaration is as follows:

```
protected_type_declaration ::=
    protected type defining_identifier [known_discriminant_part]
        is protected_definition

protected_definition ::=
    protected_operation_declaration
        [private protected_element_declaration]
        end defining_identifier ;
  | protected_operation_declaration
        private
        hidden_part ;

protected_operation_declaration ::=
    priority_pragma  entry_or_subprogram {protected_operation}

entry_or_subprogram ::=
    subprogram_declaration
  | entry_declaration ;

protected_operation ::=
    apragma
  | entry_or_subprogram

entry_declaration ::=
    entry_specification  ;
    procedure_annotation

entry_specification ::=
    entry defining_identifier [formal_part];

protected_element_declaration ::=
    variable_declaration {; variable_declaration}
```

Static semantics

1 A priority pragma (as defined in 3.3.2.1) is mandatory and must appear first[R13].

2 Only one *entry* may be declared per protected type[R1].

3 Protected elements are the data of the protected type and, if present, follow the reserved word *private* in the type declaration. All protected elements must be *statically initialized at declaration*[R16] (via a discriminant if desired). The elements are regarded as *refinement constituents* and are only visible in the protected *body*.

4 The global and derives annotation of the protected operations in the type specification use the *type name* when referring to the overall abstract state of objects of the protected type. The use of the type name in this way may be interpreted as "the particular object that is an instance of this type[R36]".

5 The global annotation may not reference any *unprotected* variables[R22]. Unprotected variables are own variables which are not declared with the modifier **`protected`** (see 3.3.3.4).

6 Protected operations may not overload names already directly visible in the enclosing package[R23].

7 A protected type must declare at least one operation[R15].

Examples

A simple protected store:

```
protected type Store
is
   pragma Priority (10);

   function Get return Integer;
   --# global Store;

   procedure Put (X : in Integer);
   --# global out Store;
   --# derives Store from X;

private
   TheStoredData : Integer := 0;
end Store;
```

An accumulator, with a count of how many times it has been called. The initial value of the accumulator and its priority are set by discriminant.

```
protected type Accumulator (Pr      : System.Any_Priority;
                            InitVal : Integer)
is
   pragma Priority (Pr);

   function Get return Integer; --returns TheStoredData
   --# global Accumulator;

   procedure Clear; --sets CallCount to 0 and TheStoredData to InitVal
   --# global out Accumulator;
   --# derives Accumulator from ;
```

```
procedure IncBy (X : in Integer); --inc CallCount and add X to TheStoredData
   --# global in out Accumulator;
   --# derives Accumulator from Accumulator, X;

private
   CallCount     : Integer := 0;
   TheStoredData : Integer := InitVal;
end Accumulator;
```

A protected type with an entry.

```
protected type Wait
is
   pragma Priority (10);

   entry GetDataWhenReady (X : out Integer);
   --# global in out Wait;
   --# derives X, Wait from Wait;

   procedure Release; -- this procedure lifts the barrier allowing access to the entry
   --# global in out Wait;
   --# derives Wait from Wait;
   ...                -- other operations
private
   TheGuard : Boolean := False; -- simple Boolean barrier used by entry
   State    : Integer := 0;     -- other data
end Wait;
```

3.3.3.2 Interrupt handlers

A protected type may include operations that are interrupt handlers. These operations, which take the form of parameterless procedures, are not, in RavenSPARK, callable in the normal sense[R24] but are instead activated by processor interrupt instructions. The operations are bound to the system of interrupts by means of pragma Attach_Handler.

Example

```
protected type Buttons
is
   pragma Interrupt_Priority (31);

   procedure Pressed;
   --# global out P.SO; -- when pressed, toggle suspension object in package P
   --# derives P.SO from ;
   pragma Attach_Handler (Pressed, -- procedure name
                          7); -- interrupt ID of type Ada.Interrupts.Interrupt_ID
end Buttons;
```

The Examiner checks that the implementation-defined parameter interrupt ID is present. If it is of an integer numeric type and a range has been defined in the configuration file (see 3.1.4.3), the Examiner will also check that the value lies in that range.

Static semantics

1 If the protected type contains a pragma Attach_Handler it must have a pragma Interrupt_Priority with an argument in the range of subtype System.Interrupt_Priority[R1].

2 Interrupt handlers must be parameterless protected procedures[R1].

3 Interrupt handlers may not be called as if they were normal procedures[R24].

4 The pragma Attach_Handler must immediately follow the procedure to which it refers[R6].

3.3.3.3 Protected subtypes

Where discriminants have been used in a protected type declaration, a named subtype that provides the missing values must be declared before any protected objects may be declared[R17].

For example, we can declare two subtypes of Accumulator with different priorities and different initial values for the accumulator.

```
subtype SmallLowAccumulator  is Accumulator (10, 0);
subtype LargeHighAccumulator is Accumulator (Pr      => 20,
                                             InitVal => 100);
```

Protected subtypes may be declared in library-level package specifications or bodies. Actual discriminants must be *static* values[R25].

3.3.3.4 Protected object declarations

A protected object declaration is a simply a variable declaration. For example:

```
TheAccumulator : SmallLowAccumulator;
```

Protected objects must be declared directly in library level packages so protected types cannot be used as part of a composite type, nor as a formal parameter type, nor as a function return type[R35].

Because a protected object is an *own variable* it must appear in its package's own variable annotation and have its properties described using the extended own variable syntax introduced in section 3.3.2.3.

Static semantics

1 A protected own variable of a user-defined protected type must:

a have its own variable declaration qualified with the modifier **`protected`**[R26];

b have a type announcement which uses its *base type*[R27];

c include a priority property[R11].

2 If the own variable is of a protected type which includes an *entry* then its property list must include the *suspendable* property[R28].

3 If the own variable is of a protected type which includes an interrupt handler then its property list must include the *interrupt* property[R27]. The interrupt property *may* also provide a list of names of user-defined *interrupt streams* which are then used in the partition-wide flow analysis. See the examples below and section 3.4.2.

4 An own variable with the *protected* modifier may not appear in a package body *refinement* annotation.

5 An own variable may have both the *protected* modifier and a *mode*. In that case it denotes a protected external variable. All the elements of such a variable must be virtual protected elements (see 3.3.3.5) and have the same mode[R29].

Examples

Given the protected object declaration:

```
TheAccumulator : SmallLowAccumulator;
```

then the following own variable declaration would be required:

```
--# own protected TheAccumulator : Accumulator (priority => 10);
```

For the declaration:

```
WaitFor : Wait;
```

we write:

```
--# own protected WaitFor : Wait (priority => 10,
--#                                suspendable);
```

And, finally, for the declaration:

```
TheButton : Buttons;
```

we have, as a minimum:

```
--# own protected TheButton : Buttons (priority => 31,
--#                                     interrupt);
```

Note that the priority property is always introduced by the *priority* keyword even if the priority is set by pragma Interrupt_Priority.

If we wanted to enrich the partition flow analysis by providing user-selected names for the external source of the interrupts then we can extend the interrupt property with an aggregate that binds operation names to their external source:

```
--# own protected TheButton : Buttons (priority => 31,
--#                  interrupt => (Pressed => FirstFloorFire));
```

The extended form of the interrupt property list is particularly useful if a protected type declares more than one interrupt handler, since it allows the effect of each individual routine to be tracked by flow analysis. Here we can check that the *first floor fire button* has the desired effect rather than the less precise *TheButton,* which is the default name for the interrupt source (i.e. the protected object name).

3.3.3.5 Virtual protected elements

The elements of a protected type are variables that cannot be connected to the external environment with address clauses or import pragmas. This makes it difficult to create secure implementations of things such as shareable protected ports. The difficulty is compounded in RavenSPARK by the prohibition of protected objects from referencing unprotected state.

To solve these problems and to increase the flexibility of protected types, SPARK allows a protected object to make sole use of otherwise unprotected state and to provide protection for it. In effect, the unprotected variable behaves exactly as if it was a protected element of the type even though it is physically located outside it. We call such a variable a *virtual protected element* of the type. Since virtual protected elements behave exactly like protected elements it follows that they are visible only in the protected body of the type that protects them. Furthermore, like protected elements, they are regarded as refinement constituents and appear only in refined annotations of the protected body.

Virtual protected elements are introduced by the *protects* property which lists the variables concerned.

Example

We could create a type that is equivalent to Accumulator (see 3.3.3.1) but with the stored values being *outside* the protected type as follows. Note that in general this is rather a foolish thing to do because it is usually better to make the scope of data as small as possible and generally the data would be better declared inside the protected type in the form of protected elements. However, this is not always possible, for example, we may wish to send each newly-written value to some external device as well as store it in the accumulator.

```
package P
--# own CallCount, TheStoredData;
--# protected TheAccumulator : Accumulator
--#             (priority => 10,
--#              protects => (CallCount, TheStoredData));
--# initializes CallCount, TheStoredData;
is

   protected type Accumulator
   is
      pragma Priority (10);
```

```
      function Get return Integer;
      --# global Accumulator;

      procedure Clear;
      --# global out Accumulator;
      --# derives Accumulator from ;

      procedure IncBy (X : in Integer);
      --# global in out Accumulator;
      --# derives Accumulator from Accumulator, X;
   end Accumulator;
...
```

Here we have declared two own variables `CallCount` and `TheStoredData` which are initialized. We then declare `TheAccumulator` of type `Accumulator` and announce in its property list that it protects (i.e. is the sole user of) `CallCount` and `TheStoredData`. Because these are normal variables (not protected elements) they may be mapped to particular addresses or memory-mapped ports; however, the protects property ensures that they are accessible only via the protected body of `Accumulator`. Further examples of the use of virtual protected elements can be found in A.3.3 and A.3.4.

Static semantics

1 Virtual protected elements must be concrete own variables declared in the *same package* as the type that will protected them[R30]. As they are concrete own variables, they may not appear in a refinement clause in the package body.

2 Virtual protected elements must either appear in an *initializes* clause (in which case they must be initialized at *declaration*) or be *external variables* (i.e. have a *mode*) [R30].

3 Only *one instance* of a protected object of a type which protects a virtual element may be declared and this declaration must be in the *same package* as the type and the element [R30]. The recommended style is to declare this protected type in the private part of its package so that it is not visible externally.

4 Own variables which are protected by virtue of being virtual elements of a protected type are only visible inside the *protected body* of that type [R30].

5 If all the elements of protected type are virtual elements which are external variables with the same mode, then the protected variable itself may be given a mode. Thus a protected type PT which protected a single input port called `InPort` would be annotated[R29]:

```
--# own protected in ThePort : PT (priority => 12,
--#                                protects => InPort);
```

3.3.3.6 Suspension objects

When the ravenscar analysis profile is selected, the Examiner makes predefined package `Ada.Synchronous_Task_Control` available (see 3.1.3.2). This package declares the type

`Suspension_Object` which may be used to declare own variables which will have the protected property. Objects of this type are *suspendable* and must be declared as such in the *property list* of their own variable declaration.

The package `Ada.Synchronous_Task_Control` provides procedures to set objects of type `Suspension_Object` to True or False and to allow a task to suspend until a suspension object becomes True.

Example

The task type `ProcessWhenReady` described in 3.3.2.1 makes use of a user-defined suspension object `DataReady`. This would be an own variable with the annotation:

```
--# own protected DataReady (Suspendable);
```

A possible body for the task type might be:

```
DataReady : Ada.Synchronous_Task_Control.Suspension_Object;
...
task body ProcessWhenReady
is
begin
   loop
      -- wait until there is something to do
      Ada.Synchronous_Task_Control.Suspend_Until_True (DataReady);
      -- do it
      P.ProcessTheState;
   end loop;
end ProcessWhenReady;
```

Static semantics

1 A variable of the predefined suspension object type must be an own variable with the *protected* modifier[R28].

2 The own variable must be declared with the *suspendable* property but cannot have the *priority*, *protects* or *interrupt* properties[R1].

3 The variable must be declared directly in a library-level package. It follows that the predefined suspension object type cannot be used as part of a composite type, as a formal parameter type in a user-defined subprogram or as a function return type[R35].

4 The variable cannot be initialized at declaration since suspension objects have a default initialization to False[R1].

3.3.3.7 Pragma Atomic types

The final form of shareable protected state is the declaration of variables of a type marked as atomic by use of pragma Atomic. Variables of such atomic types are shareable and may therefore be declared in

own variable declarations with the protected modifier. They do not need to be type announced and do not have any properties.

Example

```
package P
--# own protected State;
is
   type MyInt is range Integer'First .. Integer'Last;
   pragma Atomic (MyInt);

   State : MyInt := 0;
end P;
```

Here we show the type and variable being declared in the package specification; it might equally well be placed in the package body and suitable subprograms used to access it. It would still be shareable because the global annotations of these subprograms would reveal the fact they operated on an own variable with the *protected* property.

Static semantics

1 pragma Atomic may only be applied to user-defined, *scalar* types[R31] or to a non-tagged record type that has a single field that is a predefined type.[R37]. Since the pragma has semantic significance then care should be taken to only use it where it is clear from the compiler's documentation that the type really is atomic. (A compiler should reject invalid use but the Examiner clearly cannot).

2 A variable of an atomic type *may* be declared with the own variable modifier **`protected`**[R26]. Note that it does not *have to* be so declared and if it is not then it is treated as a normal, unprotected own variable.

3 If the **`protected`** modifier is used then the variable must be *initialized at declaration*[R16]; however, since *all* protected state is initialized at declaration it must not appear in the package's *initializes* annotation.

3.3.3.8 Protected bodies

A protected body is required for each protected type. It provides the implementation of the operations declared in the protected type. A protected body may appear in a library package body or may be a subunit; in the latter case a suitable stub appears in the package body. Protected operations may make nested calls to other protected operations subject to the static semantic restrictions below.

Syntax

The syntax of a protected body is as follows.

protected_body ::=
protected body *defining*_identifier **is**
protected_operation_item {protected_operation_item}
end *defining*_identifier ;

protected_operation_item ::=
subprogram_body | entry_body

entry_body ::=
entry_specification **when** *component*_identifier
procedure_annotation
is subprogram_implementation

and for a protected body stub:

protected_body_stub ::=
protected body *defining*_identifier **is separate** ;

Static semantics

1 Where the abstract annotation of an operation mentions the type name, a second annotation is required on the operation body; this second annotation is expressed in terms of the type's protected elements[R19] (including virtual protected elements) and obeys exactly the same rules as second annotations involving own variable refinement constituents in sequential SPARK.

2 The entry barrier (introduced by the reserved word **when**) must be a Boolean element of the protected type[R1].

3 The sequence of statements of a protected operation may not include a delay statement, a protected entry call or a call to Ada.Synchronous_Task_Control.Suspend_Until_True [R1].

4 The sequence of statements of a protected operation may not include a procedure call statement to a procedure which either *delays* or *suspends* [R1]. (A procedure's *declare* annotation indicates whether either of these applies, see 3.3.1 and 3.3.3.11).

5 A protected function body may not make a call to a protected procedure [R1]; this is an Ada rule designed to prevent side effects in protected functions. With the protection provided by SPARK, it would be possible safely to allow such procedure calls provided no side effect resulted; unfortunately, the resulting programs would no longer be legal Ada!

6 If a protected operation calls another protected operation, then the callee protected object must have the same or greater ceiling priority [R1]. (See 2.2.1.2).

Examples

A suitable body for protected type `Accumulator` as defined in section 3.3.3.1 might be:

```
protected body Accumulator
is
   function Get return Integer
   --# global TheStoredData; -- second, refined annotation
   is
   begin
      return TheStoredData;
   end Get;
```

```
   procedure Clear
   --# global out CallCount, TheStoredData;
   --# derives CallCount, TheStoredData from ;
   is
   begin
     CallCount := 0;
     TheStoredData := InitVal;
   end Clear;

   procedure IncBy (X : in Integer)
   --# global in out CallCount, TheStoredData;
   --# derives CallCount     from * &
   --#         TheStoredData from *, X;
   is
   begin
     CallCount := CallCount + 1;
     TheStoredData := TheStoredData + X;
   end IncBy;
end Accumulator;
```

A suitable body for protected type `Wait` defined in section 3.3.3.1 might be:

```
protected body Wait
is
   entry GetDataWhenReady (X : out Integer) when TheGuard
   --# global in State; out TheGuard;
   --# derives X        from State &
   --#         TheGuard from ;
   is
   begin
     X := State;
     TheGuard := False; -- reset barrier to stop any further access to State
   end GetDataWhenReady;

   procedure Release -- this procedure lifts the barrier allowing access to the entry
   --# global out TheGuard;
   --# derives TheGuard from ;
   is
   begin
     TheGuard := True;
   end Release;
   ...                  -- other operations
end Wait;
```

3.3.3.9 Use of protected variables

There are strict limitations on the way in which protected own variables may be accessed.

1 Via a subprogram or entry call where the protected object is a *global* of the called subprogram. This applies to any kind of protected variable. It is the *only* way in which an object of a user-defined

protected type may be manipulated. Where the subprogram concerned is a function, it may only appear directly in an assignment or return statement, not in a general expression[R32].

2 Via a call to a predefined operation in Ada.Synchronous_Task_Control (see 3.1.3.2). This applies only to suspension objects which are the only form of protected state that may be used as actual parameters.

3 Directly in an assignment or return statement. This applies only to a variable of a pragma Atomic type. Note that this prevents their use in general expressions [R32] or their passing as parameters.

Within a protected body, its own protected elements (including virtual protected elements) behave exactly like variables in sequential SPARK. No special rules apply at all[R33].

3.3.3.10 Protected subprogram calls

Calls to protected operations (other than to operations within the same protected body) take the syntactic form:

{package_prefix**.**}protected_variable_name**.**operation **[**(actual_parameter)**];**

For flow analysis purposes, the Examiner substitutes the name of the protected object (given by protected_variable_name in the grammar above) for the protected type name in the called operation's annotation. For example, given:

```
protected type PT
...
   procedure Modify;
   --# global in out PT; -- PT is a placeholder for the actual object being used
   --# derives PT from PT;
...
PO : PT;
...
PO.Modify; -- treated as having the flow relation --# derives PO from PO;
```

3.3.3.11 Renaming of protected operations

SPARK only permits the removal of package prefixes in a subprogram renaming. Since part of the prefix of a protected operation call is an *object* rather than a *package* it follows that protected operations *cannot* be renamed in RavenSPARK.

3.3.4 Potentially blocking operations

The Ravenscar Profile defines calls from *protected* sequences of statements to subprograms which *suspend* or *delay* as run-time exceptions. Such suspension would cause a re-schedule that undermines the enforcement of mutual exclusion via continued execution on a monoprocessor as well as contradicting assumptions made by timing analysis models. To enable detection of this error condition, RavenSPARK prohibits *functions* from suspending or delaying[R9] and requires *procedures* that *may* suspend or delay to be annotated with a suitable property declaring that they do so[R10,R12].

Examples

A procedure which waits until the absolute time passed into it as a parameter would be declared thus:

```
procedure Wait (X : in Ada.Real_Time.Time);
--# derives null from X; -- only time is affected
--# declare delay; -- declare that the procedure performs a delay
```

A procedure which performs a suspend until true on suspension object SO would be declared:

```
procedure WaitForSO;
--# global out SO;
--# derives SO from ;          -- flow relation for a suspend_until_true call
--# declare suspends => SO; -- declare the potential suspension
```

The Examiner will check that subprograms that delay or suspend are correctly annotated if their bodies are available. Great care should be taken to annotate correctly procedures with hidden bodies or those that provide external interfaces. The delay property can be used to indicate that such procedures may block due to the performance of predefined input/output operations for example.

3.3.5 The main subprogram

In sequential SPARK, the main program provides the start point for the algorithm of the entire program and is the point from which the linkable closure of the program is calculated. A main program in the Ravenscar profile performs a rather different function since most of the functional code will be distributed amongst the various tasks of the program. The main program's context clause still performs the vital role of defining what components form the overall program but the main program itself may be very small; commonly it may just be a single null statement.

When the SPARK ravenscar analysis profile is selected an additional global and (optionally) derives annotation is required; this describes the overall information flow of the entire partition (i.e. entire program) as described in the next section.

Syntax

main_program ::=
    [inherit_clause]
    main_program_annotation
    global_definition [dependency_relation]
    subprogram_body
main_program_annotation ::= **--# main_program** ;

Static semantics

1 If the ravenscar analysis profile is selected, a partition annotation comprising of at least a moded global annotation and optionally a derives annotation must follow the main_program annotation.

2 The partition annotation must mention:

a each own variable that appears in the global annotation of each task in each package that is in the transitive closure of the main subprogram

b each own variable that appears in the global annotation of each interrupt handler in each package that is in the transitive closure of the main subprogram

c each own variable that appears in the global annotation of the environment task (i.e. the main program procedure itself)

d for each protected object that includes an interrupt handler in each package that is in the transitive closure of the main subprogram, either:

i the interrupt stream names declared in the corresponding protected own variable's interrupt property list; or

ii the name of the protected object itself if no interrupt stream names have been declared.

3 The declarative part of the main program must include a priority pragma with a literal or constant value.

4 If the main program's context clause does not include any packages which declare *task* objects, then the body of the main program must end with a plain loop containing no exit statements[R34].

Example

```
with P;              -- P is the only other package in the program closure and declares a task
--# inherit P;
--# main_program; -- main program annotation
--# global in out P.State; -- partition annotation follows main_program
--# derives P.State from P.State;
procedure Main
--# derives ; -- this is the flow relation for the environment task only
is
  pragma Priority (10); -- a priority is mandatory and must appear here
begin
   null; -- all the real work is done somewhere else, in this case by the task in package P
end Main;
```

3.4 Flow analysis

The Examiner can perform data and information analysis of Ravenscar programs in exactly the same manner as for sequential programs. The analysis of sequential parts of the program is modified to capture the essential volatility of shareable protected variables whose values can be changed by other

program threads[2] at any time. An additional partition-wide form of analysis has been added to capture the effect of all the program's concurrent entities working together.

The term *communicating protected variable* appears in the following sections. To be potentially shared between program threads a variable must have been declared with the own variable modifier **own protected**. A *communicating protected variable* is one which can cause *inter-task* communication because it can be referenced *and* updated; an own protected variable declared without a mode is the only kind of variable that meets this specification. By contrast, however, if it has a *mode*, indicating that it is connected to the external environment, then it is not considered *communicating* since its mode can only be `in` or `out` (*not* `in out`) and therefore information flow cannot occur *between* tasks that share it. For example, several tasks might share a protected out port; they are guaranteed mutual exclusion in their writes; the external environment will receive all the values written; but, one task cannot pass information to another via the shared use of the out port.

3.4.1 Thread-level flow

Flow analysis at the thread level (i.e. within the sequence of statements making up subprograms, task bodies and protected bodies) is largely unaltered from sequential SPARK. The key conceptual difference concerns the use of protected variables which may be shared. In sequential code, the imports enter a sequence of statements at the top and cannot be changed during execution of the statements; the exports then emerge at their end. If a referenced variable may be shared then it has to be considered an "import" at any point where it is read since the value obtained by the read may be provided from outside the sequence of statements concerned and may be unconnected with any values previously written to that variable in the sequence of statements.

The effect is probably best illustrated by means of a simple example. Consider a subprogram that exchanges the two parameters passed to it. To effect the swap it makes use of a communicating protected variable as temporary scratchpad (clearly a foolish design).

```
procedure Swap (X, Y : in out Integer)
--# global out T; -- T is a communicating protected variable
--# derives Y    from X &
--#         X    from Y &
--#         T    from X;
is
begin
   T := X; -- definitely puts X in T and this value may get read by another thread
   X := Y; -- no communicating variables involved, straightforward assignment
   Y := T; -- T may no longer have X in it, another thread may have updated it
end Swap;
```

Because the value of T may change at any time, the annotation given for this subprogram is incorrect. We must capture the fact that the final value of Y may depend not just on X but on whatever value T may happen to take when the final assignment statement is executed. X is only one possible value that

[2] Defined for our purposes here as "execution of a task or interrupt handler".

T may hold because other threads might have changed it between execution of the first and third statements.

We capture this effect by the following rule:

- A referenced protected variable is regarded as being imported at each point where it is referenced regardless of what updates have been made to it earlier.

The correct annotation for the, poorly-designed, swap procedure is therefore:

```
--# global in out T;
--# derives Y    from X, T &
--#         X    from Y &
--#         T    from X;
```

The additional dependence of Y on T captures the fact that although it may successfully exchange X and Y, this is not guaranteed and Y may end up with some other value obtained from some unknown value of T. When we obtain the overall flow relations for the entire partition (see next section) this additional import will allow us to bind Swap together with other operations that generate values of T.

There is one important additional rule required to provide meaningful information flow analysis of a program with shared variables:

- A sequence of statements may only include a single instance of a statement which updates a *communicating* protected variable without also referencing it.

Consider a *communicating* protected variable P with operations:

```
procedure SetTo (X : in Integer);
--# global out P;
--# derives P from X; -- updates P without referencing it

procedure Modify;
--# global in out P;
--# derives P from P;

procedure Get (X : out Integer);
--# global in P;
--# derives X from P;
```

The following sequence of statements is *illegal* under the rule just stated:

```
SetTo (10);
Modify;
SetTo (20); --!
SetTo (30); --!
Get (X);
```

because it contains multiple instances of pure updates of P. Note that in sequential SPARK the first three statements would be marked as ineffective. They are not ineffective in RavenSPARK because

another program thread might be able to read the value 10 before it became 20, or 20 before it suddenly became 30!

The restriction can perhaps be best considered analogous to SPARK's requirement that a function only has one return statement. It can also be accommodated in the same way by using local variables to store intermediate values and making a single assignment to the protected variable when the value to be written is fully calculated.

Typical Ravenscar programs should rarely have problems with these rules (or with the rather strange information flows of the Swap procedure) because protected objects are normally used to provide disciplined communication between tasks rather than as arbitrary scratchpad variables!

When constructing the information flow relations for tasks and interrupt handlers, it is useful to remember that the relation we require is the one that assumes the task will eventually run or the interrupt will eventually occur. In particular, we do not regard temporal effects such as delaying or suspending to have any significance as far as information flow analysis is concerned.

3.4.2 Program wide flow

Program-wide (more accurately partition-wide) flow analysis is performed by combining the flow relations for all the active components of the program (i.e. all the tasks and interrupt handlers) in a way that connects shared outputs of one task with inputs of another. The resulting flow relation is compared with that given in the partition annotation on the main program (see 3.3.5) and differences reported as flow errors in the usual way. As well as providing an overall view of the information flow through the entire partition, this analysis provides important protection against the inadvertent generation of an incomplete program. The failure to *with* a package containing a task does not make a program illegal, it simply becomes a different, legal program with a subset of the intended behaviour. Partition-wide flow analysis is very likely to reveal the omission since omitting a task is certain to affect the calculated partition-wide flow relation which will then not match that claimed in the partition annotation.

Program-wide information flow is calculated in the following manner.

1 For each task (identified by the own task annotation of each package in the transitive closure of the main subprogram), we add any object that the task suspends on (identified by the task's property annotation) as an import that influences the exports of that task.

2 For each interrupt handler (identified by the property list of the own protected annotation of each package in the transitive closure of the main subprogram), we add, as an import, the name of the source from which the interrupt is deemed to come. By default this is the name of the protected object that contains the interrupt handler but the property annotation allows a more descriptive, user-selected name to be used instead.

3 For the enriched annotations generated above we take their union so as to establish connections between values generated by one task and referenced by another.

4 Finally we take the transitive closure of this union because each task runs continuously and the effects of inter-task information flows may eventually propagate to all tasks that share information. For example if Task 1 derives B from A and Task 2 derives C from B and Task 3 derives D from C then taking the closure ensures that the possible influence of A on D is detected.

3.5 Conversion of existing sequential SPARK programs

Users may make their first steps with RavenSPARK by adding an interrupt handler or task to an existing sequential SPARK program. We start by showing how a sequential SPARK program can be made into a RavenSPARK program with minimal changes. The existing sequential program will presumably have some form of scheduler in the main program which will take the form of an endless loop; this meets the requirement that the environment task must end in a plain loop if there are no other tasks present. The main program will also import and export various own variables in the packages that appear in its context clause. Again this meets the requirements of RavenSPARK which permit tasks to make use of unprotected state provided it is not shared. Some of this state may be initialized at declaration and imported by the main program. This is also legal because the main subprogram, unlike other tasks, does not run until program elaboration is complete and so there is no risk of a race condition in the initialization of state that the main program imports. We can therefore convert a legal sequential SPARK program to a legal RavenSPARK one by simply adding a partition annotation which, initially at least, will be identical to the current main program annotation. In effect, we now have a tasking program comprising a single task (the environment task) whose program-wide information flow is identical to that of the main subprogram.

From this starting point we can now start to add new concurrency constructs. The process will involve:

1 Converting unprotected state to protected state where it now has to be shared.

2 Adjusting the partition (and perhaps main program) annotation to incorporate the effect of the newly-added concurrency items.

3.6 Proof support

3.6.1 Run-time checks

The generation of run time checks is completely unaffected by the selection of the ravenscar analysis profile. *We strongly recommend that these checks be generated* because an unhandled exception in a task or interrupt handler could cause difficult-to-diagnose failure modes (see 2.2.1.1). Furthermore, if an exception causes a task to terminate, the subsequent program behaviour is implementation defined.

3.6.2 Partial correctness

There is currently no support for partial correctness proofs involving communicating, protected variables. To avoid the possibility of invalid proofs being generated, SPARK prohibits potentially *communicating* protected variables from appearing in *proof annotations*. Furthermore, since tasks can

neither be called nor are they permitted to terminate, proof annotations are not permitted on task type or task body declarations.

This restriction does not prevent the proof of sequential code sequences, nor from proving that protected operation bodies have their intended behaviour in terms of the protected elements they manipulate.

For example, the protected body for Accumulator from section 3.3.3.8) could be written:

```
protected body Accumulator
is
   function Get return Integer
   --# global TheStoredData;
   --# return TheStoredData;
   is
   begin
      return TheStoredData;
   end Get;

   procedure Clear
   --# global out CallCount, TheStoredData;
   --# derives CallCount, TheStoredData from ;
   --# post CallCount = 0 and TheStoredData = InitVal;
   is
   begin
     CallCount := 0;
     TheStoredData := InitVal;
   end Clear;

   procedure IncBy (X : in Integer)
   --# global in out CallCount, TheStoredData;
   --# derives CallCount     from * &
   --#         TheStoredData from *, X;
   --# post CallCount = CallCount~ + 1 and
   --#      TheStoredData = TheStoredData~ + X;
   is
   begin
     CallCount := CallCount + 1;
     TheStoredData := TheStoredData + X;
   end IncBy;
end Accumulator;
```

and Verification Conditions (VCs) generated to show that the protected operations perform their intended (concrete) actions.

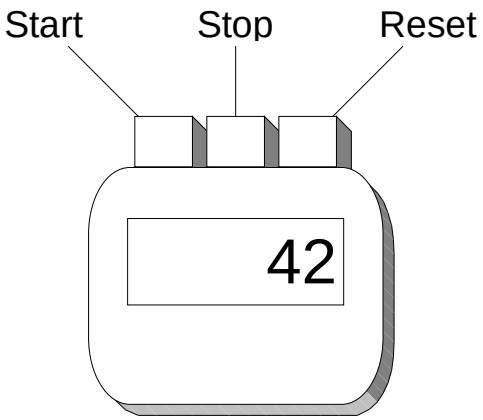
VCs may also be generated to prove properties of the code not dependent on shared objects. For example, it would be possible to prove that the final value X was the initial value of Y in the badly-designed swap algorithm (described in section 3.4.1) but not anything about the final value of Y because that depends on the *communicating* protected variable T.

4 Examples

Two examples are produced here; the first in full. They are liberally commented and serve to illustrate many of the principles and rules of RavenSPARK discussed above.

4.1 Simple stopwatch

The first example is a simple stopwatch that counts and displays seconds. The watch has three buttons: the first starts the timer, the second stops it and the third resets the counter to zero but does not change whether the watch is running or not.



The program comprises three main packages: a *User* interface, a *Timer* task and a *Display* manager. A fourth package provides a set of constants that can be used to "tune" the behaviour of the program.

```
with System, Ada.Real_Time;
--# inherit System,
--#         Ada.Real_Time;
package TuningData
is
   -- priorities
   UserPriority    : constant System.Interrupt_Priority := 31;
   TimerPriority   : constant System.Priority := 15;
   DisplayPriority : constant System.Interrupt_Priority := 31;

   -- task periodicities
   TimerPeriod     : constant Ada.Real_Time.Time_Span :=
     Ada.Real_Time.Milliseconds (1000); -- 1 second counter
end TuningData;
```

The user package contains a protected type with an interrupt handler for each of the three buttons. The start and stop button alter a suspension object that controls whether the timer task runs. The reset button calls the display manager to zero the second count.

```
with TuningData;
--# inherit Timer,
--#         Display,
--#         TuningData;
package User
--# own protected Buttons : PT -- Buttons will be declared in the package body
--#                 (Interrupt => (StartClock => StartButton,
--#                                StopClock  => StopButton,
--#                                ResetClock => ResetButton),
--#                  -- we provide interrupt stream names for the three interrupt handlers
--#                  Priority => TuningData.UserPriority);
is
private -- there is no need to make the protected type visible outside this package

   protected type PT is
      pragma Interrupt_Priority (TuningData.UserPriority);
      -- because the protected type has the interrupt property it must use pragma Interrupt_Priority

      procedure StartClock;
      --# global out Timer.Operate; -- suspension object that controls the timer
      --# derives Timer.Operate from ;
      pragma Attach_Handler (StartClock, 1);

      procedure StopClock;
      --# global out Timer.Operate;
      --# derives Timer.Operate from ;
      pragma Attach_Handler (StopClock, 2);

      procedure ResetClock;
      --# global in out Display.State;
      --# derives Display.State from *; -- reset the display to 0
      pragma Attach_Handler (ResetClock, 3);
   end PT;
end User;
```

The timer package declares the suspension object Operate and procedures to allow it to be set and reset. It also declares a periodic task that actually counts the seconds timed by the stopwatch.

```
with TuningData;
--# inherit Ada.Synchronous_Task_Control,
--#         Ada.Real_Time,
--#         Display,
--#         TuningData;
package Timer
--# own protected Operate (suspendable); -- Operate is a suspension object
--#     task TimingLoop : TT;
is
   -- These two procedures simply toggle suspension object Operate
   procedure StartClock;
   --# global out Operate;
   --# derives Operate from ;
```

```
   procedure StopClock;
   --# global out Operate;
   --# derives Operate from ;

private
   task type TT
   --# global    out Operate;
   --#        in out Display.State;
   --#        in     Ada.Real_Time.ClockTime;
   --# derives Operate       from &
   --#         Display.State from * &
   --#         null          from Ada.Real_Time.ClockTime;
   --# declare suspends => Operate;
   is
      pragma Priority (TuningData.TimerPriority);
   end TT;
end Timer;
```

Finally, the Display package; this maintains an internal counter for the seconds and also protects an output port which causes the counter to be displayed on the stopwatch's screen.

```
with TuningData;
--# inherit TuningData;
package Display
--# own out       Port; -- an output port ...
--#     protected State : PT (priority => TuningData.DisplayPriority,
--#                           protects => Port); -- which is protected by State
is
   -- these two procedures simply export the protected operations in PT.  Alternatively, we could
   -- declare object State in the package specification and make its operations directly available.
   procedure Initialize;
   --# global in out State;
   --# derives State from *;

   procedure AddSecond;
   --# global in out State;
   --# derives State from *;

private
   protected type PT is
      pragma Interrupt_Priority (TuningData.DisplayPriority);
      -- we need this surprisingly high priority because procedure Reset is called
      -- by interrupt handler User.ResetClock and we would have a priority ceiling violation
      -- if protected object State was lower than that of the caller.

      -- add 1 second to stored time and send it to port
      procedure Increment;
      --# global in out PT;
      --# derives PT from *;
```

```
      -- clear time to 0 and send it to port
      procedure Reset;
      --# global in out PT;
      --# derives PT from *;

   private
      Counter : Natural := 0; -- the second counter
   end PT;
end Display;
```

We now have enough infrastructure to write and analyse the main program or environment task that assembles these components in to a working stopwatch. We do not, yet, need bodies of the three principal packages we have written.

```
with User, Timer, Display; -- this clause determines what the program includes
--# inherit User, Timer, Display, Ada.Real_Time;
--# main_program; -- following annotation is the partition annotation
--# global in     User.StartButton, -- this is a user-defined, interrupt stream name
--#               User.StopButton,  -- this is a user-defined, interrupt stream name
--#               User.ResetButton, -- this is a user-defined, interrupt stream name
--#               Ada.Real_Time.ClockTime;
--#        in out Timer.Operate,
--#               Display.State;
--# derives Timer.Operate from *,
--#                            User.StartButton,
--#                            User.StopButton &
--#         Display.State from *,
--#                            Timer.Operate,
--#                            User.StartButton,
--#                            User.StopButton,
--#                            User.ResetButton &
--#         null          from Ada.Real_Time.ClockTime;
procedure Main
--# derives ; -- this is just the annotation for the environment task itself
is
   pragma Priority (10); -- the environment task must have a priority
begin
   null; -- all the real work is done by the interrupts and the task, not by the main program!
end Main;
-- the environment task does not need a plain loop because the program contains at least one task
```

We can check that the partition flow relation is as expected noting that all three user buttons affect the state of the display but that the reset button does not affect suspension object `Timer.Operate` and so does not start or stop the clock.

We can now code bodies for the three main packages (TuningData does not need one). The user interface is very simple with each interrupt routine simply calling an appropriate procedure in either Timer or Display.

```
with Timer, Display;
package body User
is
   Buttons : PT; -- declaration of the protected object announced in the own variable clause

   protected body PT is
      procedure StartClock
      is
      begin
         Timer.StartClock;
      end StartClock;

      procedure StopClock
      is
      begin
         Timer.StopClock;
      end StopClock;

      procedure ResetClock
      is
      begin
         Display.Initialize;
      end ResetClock;
   end PT;
end User;
```

The timer package body is as follows:

```
with Ada.Synchronous_Task_Control,
  Ada.Real_Time,
  Display;
use type Ada.Real_Time.Time;
package body Timer
is
   Operate    : Ada.Synchronous_Task_Control.Suspension_Object;
   TimingLoop : TT;

   procedure StartClock
   is
   begin
      Ada.Synchronous_Task_Control.Set_True (Operate);
   end StartClock;

   procedure StopClock
   is
   begin
      Ada.Synchronous_Task_Control.Set_False (Operate);
   end StopClock;
```

```
   task body TT is
      Release_Time : Ada.Real_Time.Time;
      Period : constant Ada.Real_Time.Time_Span :=
        TuningData.TimerPeriod;
   begin
      Display.Initialize; -- ensure we get 0 on the screen at start up
      loop
         -- wait until user allows clock to run
         Ada.Synchronous_Task_Control.Suspend_Until_True (Operate);
         -- Keep the task running, the previous call will have set Operate to False.
         -- Note that this is rather questionable - we risk a race condition with the operation of
         -- the user's stop button. More robust solutions are left as an exercise for the reader!
         Ada.Synchronous_Task_Control.Set_True (Operate);
         -- once running, count the seconds
         Release_Time := Ada.Real_Time.Clock; -- must be simple assignment
         Release_Time := Release_Time + Period;
         delay until Release_Time;
         -- each time round, update the display
         Display.AddSecond;
      end loop;
   end TT;
end Timer;
```

Finally, the Display; this is of interest because of its declaration of a memory-mapped port which is protected by the protected object State.

```
package body Display
is
   State : PT;
   Port  : Integer;
   for Port'Address use 16#FFFF_FFFF#;

   protected body PT is
      procedure Increment
      --# global in out Counter; out Port; -- refined annotation
      --# derives Port, Counter from Counter;
      is
      begin
         Counter := Counter + 1;
         Port := Counter;
      end Increment;

      procedure Reset
      --# global out Counter, Port;
      --# derives Counter, Port from ;
      is
      begin
         Counter := 0;
         Port := Counter;
      end Reset;
   end PT;
```

```
      -- these subprogram bodies must follow the body of PT since they call operations in it
      procedure Initialize
      is
      begin
         State.Reset; -- protected operation call
      end Initialize;

      procedure AddSecond
      is
      begin
         State.Increment;
      end AddSecond;
   end Display;
```

4.2 Mine pump

The mine pump example is taken from [3]. The basic functional requirement of the mine pump is that it pumps water out of a mine when the water level gets too high. For safety reasons the pump must not operate when the methane level is too high. Also for safety reasons the operator must be able to manually switch the pump on and off. The operator must be kept informed of the system state and all critical events must be recorded in a log.

A fully worked implementation of this larger example can be found in [5] which forms part of the Examiner manual set.

A Appendix: RavenSPARK templates and building blocks

In this appendix we provide templates for commonly-occurring program elements. These serve to reinforce the language description given earlier and act as a source of reference material.

A.1 Periodic task

A.1.1 Overview

A periodic task is a task that runs at set intervals. The intervals are controlled by a *delay until* statement that according to the rules of the Ravenscar profile, must have an absolute (not relative) time as its argument. The initial time can be obtained from the Ada.Real_Time.Clock or, more usually, from some program-wide start time provided by an "epoch" package; see A.4.

A.1.2 Template

The specification of a periodic task takes the form:

```
task type T <any discriminants go here>
--# global ...;
--# derives ...; -- describe the effect of repeated execution of the task body
is
   pragma Priority (...); -- or Interrupt_Priority
end T;
```

and the body

```
task body T
is
   -- Obtain first time to run from package Epoch. Less usually, we could ask the task
   -- to run as soon as possible by setting the first Release_Time to Ada.Real_Time.Clock
   -- in the initialization code.  In this case, the annotation of the task type would have
   -- to import the own variable Ada_Real_Time.ClockTime, probably deriving null
   -- from it.
   Release_Time: Ada.Real_Time.Time:= Epoch.T_Start;
   -- Setting the period to 50ms makes use of a predefined function of Ada.Real_Time
   Period : Ada.Real_Time.Time_Span :=
              Ada.Real_Time.Milliseconds(50);
```

```
begin
   <initialisation code>
   loop
      delay until Release_Time; -- deterministic release
      -- perform the periodic action required
      Do_Periodic_Work;
      -- calculate next time to run
      Release_Time := Release_Time + Period;
   end loop;
end T;
```

A.2 Sporadic task

A sporadic task is a task that is released by some external stimulus rather than by the passing of time.

A.2.1 Release by suspension object

A.2.1.1 Overview

The simplest form of sporadic task is one that is released, without data transfer, by a predefined suspension object being set to True by some other program thread. The suspension object can be seen as a kind of flag or semaphore on which the sporadic task waits.

A.2.1.2 Template

The specification of a sporadic task is very similar to that given for a periodic task above; however, it will include a declare annotation giving the name of the object on which the task suspends. First we will need the suspension object to be declared somewhere. It is a protected own variable thus:

```
package P
--# own task TheTask : T;
--#     protected T_SO (suspendable);
-- T_SO will be of type Ada.Asynchronous_Task_Control.Suspension_Object
is ...
```

The task type can then be declared:

```
task type T <any discriminants go here>
--# global ...;
--# derives ...; -- describe the effect of repeated execution of the task body
--# declare suspends => T_SO; -- admit task's suspension behaviour
is
   pragma Priority (...); -- or Interrupt_Priority
end T;
```

Note that the suspension object will just be an export from the task because all the operations of Ada.Synchronous_Task_Control are defined as just exporting the suspension object. Note that the

suspension object is *not* regarded as affecting any of the task's other exports since it only affects when the *task* runs; it is *not* therefore ever an *import* to the task.

A typical body for a task that suspends on a suspension object has the following form.

```
…
task body T is
begin
   loop
      Ada.Synchronous_Task_Control.Suspend_Until_True (T_SO);
      Do_Aperiodic_Work;
   end loop;
end T;
```

The task will run when some other program thread executes the statement:

```
Ada.Synchronous_Task_Control.Set_True (P.T_SO);
```

Since the release of the task by the suspend statement also resets T_SO to False the task will execute just once and suspend again unless some other process sets T_SO to True during the task body loop's execution.

A.2.1.3 Example

A common idiom is to place the task type, task object and the object on which it suspends in a single package and for that package to export procedures to start (and also, perhaps, stop) the task. For example.

```
--# inherit Ada.Synchronous_Task_Control;
package P
--# own protected SO (suspendable);
--#     task       T : TT;
--#                LocalState;
is
   procedure StartTask;
   --# global out SO;
   --# derives SO from ;

private
   task type TT
   --# global    out LocalState,  -- may not import unprotected LocalState
   --#                 SO;
   --# derives LocalState,
   --#         SO         from ;
   --# declare suspends => SO;
   is
      pragma Priority (10);
   end TT;
end P;

with Ada.Synchronous_Task_Control;
package body P
is
```

```
   SO : Ada.Synchronous_Task_Control.Suspension_Object;
   T  : TT;
   LocalState : Integer;

   procedure StartTask
   is
   begin
      Ada.Synchronous_Task_Control.Set_True (SO);
   end StartTask;

   task body TT is
   begin
      LocalState := 0; -- a task is responsible for initializing unprotected state it uses
      loop
         Ada.Synchronous_Task_Control.Suspend_Until_True (SO);
         LocalState := LocalState + 1;
      end loop;
   end TT;
end P;
```

A.2.2 Release by entry

A.2.2.1 Overview

As an alternative to suspending on a suspension object, a task may suspend on an *entry* call to a protected object. A call to an entry is very similar to a subprogram call except that a task may suspend until some barrier in the body of the entry is lifted. Suspending on an entry provides a richer set of options than using a suspension object because it allows data transfer to take place. A task may wait until a value is available or wait until a protected object is willing to accept some data and then effect the transfer via the entry call.

A.2.2.2 Template

First we need a protected object with an entry. For this example, we place the protected object and its type declaration in the same package.

```
package Q
--# own protected PO : PT (priority => 10, suspendable);
is
   type Data is ...;

   protected type PT is
      pragma Priority (10);
```

```
      procedure Signal (D: in Data);
      --# global in out PT;
      --# derives PT from PT, D;

      entry Wait       (D: out Data);
      --# global in PT;
      --# derives D from PT;

   private
      TheData    : Data    := ...;
      TheBarrier : Boolean := False; -- entry barrier must be a protected element
   end PT;

   PO : PT;

end Q;
```

Note that the protected object PO is declared with the property suspendable because it contains an entry. For simplicity, we declare the object PO in the specification of package P. More typically it would be declared in the body and exported subprograms used to make its protected operations available outside the package.

We can now declare the task. The specification will be similar to that for a task that suspends on a suspension object (see A.2.1.2) except that it will name the protected object PO in its *suspends* property.

The task body will take the form:

```
task body T is
   My_Data : P.Data;
begin
   loop
      Q.PO.Wait (My_Data); -- if necessary suspend until data becomes available
      Operate_On (My_Data);
   end loop;
end T;
```

The barrier preventing completion of the call to `Wait` will be lifted when some other thread calls

```
Q.PO.Signal (SomeDataValue);
```

which passes some data into PO and releases the task suspended on `Wait` to use these data. Typically the last action of `Wait` will be to close the barrier again causing the task to suspend again until new data become available. An example of a protected body implementing an entry can be found in section A.3.4).

A.3 Protected object

The general form of protected objects of user-defined protected types has been extensively covered in section 3.3.3.1. In this appendix we introduce examples of particular useful forms of protected object.

A.3.1 A protected data structure

We take as an example a protected stack. This is a familiar LIFO stack but one which can be shared between program threads. This sharing imposes some interesting design constraints. A typical unshared stack might provide operations to clear the stack; push or pop values; and test whether the stack is empty or not. Some of these make less sense in a shared environment. For example, clearing the stack followed immediately by pushing a value on to it does not guarantee that we have a stack with a single value in it because another task might gain control and push (or, more dangerously, pop) between the clear and the push statements. Similarly, checking that the stack is not empty before popping it provides no protection from another thread emptying the stack just before the pop operation is executed. The solutions to these design problems depend on the intended use of the stack. For this example we make the following choices:

1 The stack is initialized ready for use when instantiated (this is an inherent property of sharable protected state in RavenSPARK).

2 A ClearAndPush operation is provided rather than separate Clear and Push operations; this ensures that we have an atomic way of generating a stack with a single entry that we have just added.

3 The Pop operation is an entry. A task trying to pop an empty stack will suspend until there is something there to pop.

4 Attempts to push onto a full stack are simply ignored and the pushed value is lost.

```
with System; -- package System is provided via the configuration file mechanism
--# inherit System;
package Stack
--# own protected State : StackType (priority => 1, suspendable);
is
   -- exported procedures to make available the operations of object State
   procedure Push (X : in Integer);
   --# global in out State;
   --# derives State from *, X;

   procedure ClearAndPush (X : in Integer);
   --# global out State;
   --# derives State from X;

   procedure Pop (X : out Integer);
   --# global in out State;
   --# derives X, State from State;
   --# declare suspends => State; -- needed because body calls an entry in State

private
   -- ideally these type declarations would be inside protected type StackType but that's not Ada
   MaxDepth : constant := 100;
   type Ptrs is range 0 .. MaxDepth;
   subtype Entries is Ptrs range 1 .. MaxDepth;
   type Vectors is array (Entries) of Integer;
```

```
   protected type StackType (Pr : System.Priority)
   is
      pragma Priority (Pr);

      procedure SPush (X : in Integer);
      --# global in out StackType;
      --# derives StackType from *, X;

      procedure SClearAndPush (X : in Integer);
      --# global out StackType;
      --# derives StackType from X;

      entry SPop (X : out Integer);
      --# global in out StackType;
      --# derives X, StackType from StackType;

   private -- the protected elements of StackType
      Ptr      : Ptrs := 0;
      Vector   : Vectors := Vectors'(Entries => 0);
      NotEmpty : Boolean := False;
      -- these static initializations give us an empty stack to start with
   end StackType;
end Stack;

package body Stack
is
   subtype SlowStack is StackType (1); -- priority set by actual discriminant
   State : SlowStack; -- declare the own variable

   protected body StackType is
      procedure SPush (X : in Integer)
      --# global in out Ptr, Vector; -- refined annotation in terms of elements
      --#           out NotEmpty;
      --# derives Ptr from Ptr &
      --#         Vector   from Vector, Ptr, X &
      --#         NotEmpty from ;
      is
      begin
         if Ptr < MaxDepth then -- guard against push on full stack
            Ptr := Ptr + 1;
            Vector (Ptr) := X;
         end if;
         NotEmpty := True;
      end SPush;

      procedure SClearAndPush (X : in Integer)
      --# global out Ptr, Vector, NotEmpty;
      --# derives Ptr, NotEmpty from &
      --#         Vector        from X;
      is
      begin
         Ptr := 1;
```

```
            NotEmpty := True;
            Vector := Vectors'(1 => X, others => 0);
         end SClearAndPush;

         entry SPop (X : out Integer) when NotEmpty -- entry barrier
         --# global in Vector; in out Ptr; out NotEmpty;
         --# derives X             from Vector, Ptr &
         --#         NotEmpty, Ptr from Ptr;
         is
         begin
            -- since the call is guarded by NotEmpty we know that Ptr > 0[3]
            X := Vector (Ptr);
            Ptr := Ptr - 1;
            NotEmpty := Ptr > 0;
         end Spop;
      end StackType;

      -- exported procedures must follow protected body

      procedure Push (X : in Integer)
      is
      begin
         State.SPush (X); -- each procedure simply calls the matching protected operation
      end Push;

      procedure ClearAndPush (X : in Integer)
      is
      begin
         State.SClearAndPush (X);
      end ClearAndPush;

      procedure Pop (X : out Integer)
      is
      begin
         State.SPop (X);
      end Pop;
   end Stack;
```

Note that because SPARK strictly prohibits overloading and the generation of “holes in scope”, we cannot use the same name for the protected operations and the subprograms that export them.

A.3.2 Interrupt handler

An interrupt handler is parameterless protected procedure which is executed not by a procedure call statement but by an external event signalled by an interrupt.

[3] Note that Release 7.0 of the SPARK Examiner does not automatically include information derived from the guard expression in the run-time checks that it generates.

For the illustrative example we have an interrupt handler that pushes the value 1 onto the protected stack (defined in section A.3.1); the effect of this might be to release a task that is currently waiting on the Pop entry.

```
--# inherit Stack; -- from A.3.1
package Interrupts
--# own protected Handler : PT
--#   (priority => 31, -- must be in the range System.Interrupt_Priority
--#    interrupt => (Event => TypeOneInterrupt)); -- user-supplied name
is
private
   protected type PT is
      pragma Interrupt_Priority (31);

      procedure Event;
      --# global in out Stack.State;
      --# derives Stack.State from Stack.State;
      pragma Attach_Handler (Event, 42); -- make it a handler
   end PT; -- no protected elements declared
end Interrupts;

package body Interrupts
is
   Handler : PT;

   protected body PT is separate; -- just for illustrative purposes
end Interrupts;

with Stack;
separate (Interrupts)
protected body PT is
   procedure Event
   is
   begin
      Stack.Push (1);
   end Event;
end PT;
```

As an interesting aside, this example will not actually work with the stack defined in section A.3.1 because the interrupt handler has a priority of 31 and the stack a ceiling priority of 1. The rules of the Ravenscar Profile prohibit protected calls to have a *decreasing* priority value. To enable the code to work, we would need to raise the ceiling priority of the stack to at least 31.

A.3.3 Thread-safe I/O port

This example takes the form of a memory-mapped input port that can be safely shared between several program threads. Each caller who polls the port is guaranteed uninterrupted access to it. The implementation makes use of virtual protected elements.

```
package SharedPort
--# own           in RawPort; -- unprotected memory-mapped port
--#     protected in SafePort : PortType
--#        (priority => 10,
--#         protects => RawPort); -- guarantees sole ownership of RawPort
-- Note that we can declare SafePort with mode in because all its elements (RawPort) are mode in
is
   function Read return Natural;
   --# global SafePort;

private
   protected type PortType is
      pragma Priority (10);

      function PRead return Natural;
      --# global PortType;
   end PortType; -- no actual elements, only the virtual element RawPort
end SharedPort;

package body SharedPort
is
   RawPort : Natural;
   for RawPort'Address use 16#FFFF_FFFF#; -- connect it to the environment
   pragma Volatile (RawPort); -- stop the compiler optimising the port away!

   SafePort : PortType;

   protected body PortType is
      function PRead return Natural
      --# global RawPort;
      is
         ReadLocal : Natural;
      begin
         ReadLocal := RawPort;
         if not ReadLocal'Valid then
            ReadLocal := 0; -- "safe" value
         end if;
         return ReadLocal;
      end PRead;
   end PortType;
```

```
   function Read return Natural
   is
   begin
      return SafePort.PRead;
   end Read;
end SharedPort;
```

A.3.4 Interrupt-driven input port

The thread-safe port in the previous section allows a number of users to poll the port without interfering with each other; however, polling can be wasteful of resources and it is often better for a consuming task to suspend until data becomes available. The interrupt-driven port that follows provides the required behaviour because a task can suspend on the exported Read procedure until an interrupt signals that data is available to read.

```
package InterruptPort
--# own           in RawPort;
--#     protected    SafePort : PortType
--#        (priority => 31, -- must be in range System.Interrupt_Priority
--#         protects => RawPort, -- has sole access to this port
--#         interrupt, -- contains an interrupt handler, we choose not to provide a name
--#         suspendable); -- because it contains an entry
-- Note that we cannot make SafePort "protected in" because not all of its elements are of mode in
is
   -- the external user interface for reading the port
   procedure Read (X : out Natural);
   --# global in out SafePort;
   --# derives X, SafePort from SafePort;
   --# declare suspends => SafePort; -- because it calls the entry

private
   protected type PortType is
      pragma Interrupt_Priority (31);

      procedure DataReady;
      --# global in out PortType;
      --# derives PortType from PortType;
      pragma Attach_Handler (DataReady, 5); -- make procedure a handler

      entry PRead (X : out Natural);
      --# global in out PortType;
      --# derives X, PortType from PortType;
   private
      DataIsReady : Boolean := False;
      TheData     : Natural := 0;
   end PortType;
end InterruptPort;
```

```
package body InterruptPort
is
   RawPort : Natural;
   for RawPort'Address use 16#FFFF_FFFF#;
   pragma Volatile (RawPort); -- stop the compiler optimising the port away!

   SafePort : PortType;

   protected body PortType is
      -- this procedure gets executed by interrupt when data is ready at RawPort
      procedure DataReady
      --# global out DataIsReady, TheData; in RawPort;
      --# derives DataIsReady from &
      --#         TheData     from RawPort;
      is
         ReadLocal : Natural;
      begin
         ReadLocal := RawPort;
         if not ReadLocal'Valid then
            ReadLocal := 0; -- provide "safe" value if data corrupt
         end if;
         TheData := ReadLocal;
         DataIsReady := True; -- open the barrier to allow data to be read
      end DataReady;

      entry PRead (X : out Natural) when DataIsReady
      --# global out DataIsReady; in TheData;
      --# derives DataIsReady from &
      --#         X           from TheData;
      is
      begin
         X := TheData;
         DataIsReady := False; -- don't let data get read more than once
      end PRead;
   end PortType;

   procedure Read (X : out Natural)
   is
   begin
      SafePort.PRead (X);
   end Read;
end InterruptPort;
```

A.4 Co-ordinated task start up

A common paradigm of multi-tasking programs is to set the initial release time of each periodic task to a fixed offset from a baseline start time. By default, each task is free to start running as soon as it is activated. The easiest way of achieving this is to declare a package providing constants of type Ada.Real_Time.Time which will be the start times of the various tasks. The same package can conveniently declare the periodicities of the tasks as well. Common terminology is to call the baseline

start time from which everything else is measured as the "Epoch" and a package of this name is a good place to store the various time constants needed.

To avoid cluttering the annotations of tasks with information concerning only their initial start time, we recommend declaring the various epoch-related times as *constants*. To allow this, RavenSPARK, as noted in section 3.1.3.1, allows the use of predefined functions for constant initialization as follows:

1 The function `Ada.Real_Time.Clock` may be used to initialize a *constant directly in* a *library level package*.

2 The other predefined functions of package `Ada.Real_Time` such as `Milliseconds`, may be used in *elaboration context*, i.e. they may be used to initialize variables and constants.

Using these relaxations we can conveniently generate an Epoch package which has the typical form:

```
with Ada.Real_Time;
use type Ada.Real_Time.Time;
--# inherit Ada.Real_Time;
package Epoch
is
   StartTime  : constant Ada.Real_Time.Time := Ada.Real_Time.Clock;

   StartTask1 : constant Ada.Real_Time.Time := StartTime +
                           Ada.Real_Time.Milliseconds (10);
   StartTask2 : constant Ada.Real_Time.Time := StartTime +
                           Ada.Real_Time.Milliseconds (20);
   StartTask3 : constant Ada.Real_Time.Time := StartTime +
                           Ada.Real_Time.Milliseconds (30);

   Task1Period : constant Ada.Real_Time.Time_Span :=
                            Ada.Real_Time.Milliseconds (7);
   Task2Period : constant Ada.Real_Time.Time_Span :=
                            Ada.Real_Time.Milliseconds (19);
   Task3Period : constant Ada.Real_Time.Time_Span :=
                            Ada.Real_Time.Milliseconds (23);

end Epoch;
```

The body of periodic task Task1 would then take the form:

```
task body Task1
is
   Release_Time: Ada.Real_Time.Time:= Epoch.StartTask1;
begin
   loop
      delay until Release_Time; -- co-ordinated release with other tasks
      -- perform the periodic action required
      Do_Periodic_Work;
      -- calculate next time to run
      Release_Time := Release_Time + Epoch.Task1Period;
   end loop;
end T;
```

B Syntax summary

B.1 Package annotations

own_variable_clause ::=
--# own own_variable_specification {own_variable_specification}

own_variable_specification ::=
own_variable_list [**::=** type_mark] [**(**property_list**)] ;**

own_variable_list ::=
own_variable_modifier own_variable { **,** own_variable_modifier own_variable }

own_variable_modifier ::= mode | **task** | **protected** | **protected in** | **protected out**

B.2 Delay statement

delay_statement ::= **delay until** *time*_expression

B.3 Task type declaration

task_type_declaration ::=
task type *defining*_identifier [known_discriminant_part] task_type_annotation task_definition

task_type_annotation ::=
moded_global_definition [dependency_relation] [declare_annotation]

task_definition ::=
is priority_pragma {*apragma*} **end** *defining*_identifier **;**

priority_pragma ::=
pragma Priority **(**expression**);** |
pragma Interrupt_Priority **(**expression**);**

known_discriminant_part ::=
(discriminant_specification {**;** discriminant_specification})

discriminant_specification ::=
identifier_list **:** type_mark

declare_annotation ::=
--# declare property_list **;**

property_list ::= property {**,** property}

property ::=
name_property
| name_value_property

name_property ::=
delay
| identifier

name_value_property ::=
identifier **=>** aggregate | expression

B.4 Task body declarations

task_body ::=
task body *definining*_identifier procedure_annotation **is**
subprogram_implementation **;**

task_body_stub ::=
task body *definining*_identifier procedure_annotation **is separate ;**

B.5 Protected type declaration

protected_type_declaration ::=
protected type *defining*_identifier [known_discriminant_part]
is protected_definition

protected_definition ::=
protected_operation_declaration
[**private** protected_element_declaration]
end *defining*_identifier ;
| protected_operation_declaration
private
hidden_part **;**

protected_operation_declaration ::=
priority_pragma entry_or_subprogram {protected_operation}

entry_or_subprogram ::=
subprogram_declaration
| entry_declaration ;

protected_operation ::=
apragma
| entry_or_subprogram

entry_declaration ::=
entry_specification ;
procedure_annotation

entry_specification ::=
entry *defining*_identifier [formal_part];

protected_element_declaration ::=
variable_declaration {**;** variable_declaration}

B.6 Protected body declaration

protected_body ::=
protected body *defining*_identifier **is**
protected_operation_item {protected_operation_item}
end *defining*_identifier ;

protected_operation_item ::=
subprogram_body | entry_body

entry_body ::=
entry_specification **when** *component*_identifier
procedure_annotation
is subprogram_implementation

protected_body_stub ::=
protected body *defining*_identifier **is separate** ;

B.7 Protected operation call

This form of operation call is used to call a protected operation from outside its protected body. Calls from within its body use normal subprogram calling syntax.

{*package*_prefix**.**}*protected_variable*_name**.**operation **[**(actual_parameter)**];**

An entry call is similar to any other operation call:

entry_call_statement ::= *entry*_name [actual_parameter_part] ;

B.8 Main program annotation

main_program ::=
[inherit_clause]
main_program_annotation
global_definition [dependency_relation]
subprogram_body
main_program_annotation ::= **- -# main_program** ;

C RavenSPARK design rationale

This section contain notes clarifying why certain RavenSPARK design decision were made. Each is identifed with the letter R and a number and these are used as endnote indicators in the main body of the document.

Identifier	Rationale
R1	A Ravenscar profile rule; the rationale for these can be found in [1]. The RavenSPARK rule is either directly required by the Ravenscar profile rules or omits something from RavenSPARK because it is not useful in the context of the Ravenscar profile.
R2	Since each call to `Ada.Real_Time.Clock` returns a potentially different value it does not make sense to use it to co-ordinate task start up other than via a constant. The value in the "epoch" constant is a true constant in the sense that its value is fixed for the life of the program; however, its value has been obtained from the external clock. We can never know or make use of the absolute value of the constant but that does not stop us starting tasks at "epoch + some_time_period". Ultimately this rather un-SPARK like use of an external function is allowed because it is too useful to prohibit! The restriction to initializing a constant mitigates the problems associated with using an external function in elaboration code.
R3	The reason SPARK prohibits the use of user-defined functions in elaboration code is to avoid elaboration order dependencies. Since the time conversion functions are predefined, this is not a problem and they can be safely used as long as their arguments are static.

R4	The signature of `Ada.Synchronous_Task_Control.Set_True` (and `Set_False`, `Suspend_Until_True`) and their annotations do not match. Indeed, it would not be possible to process these declarations as SPARK source because the Examiner would report the mismatch and require the procedure parameters to be imports because they are of mode in out. However, the referencing of the procedure parameter is solely for purposes inside the Ada run-time library and outside the boundary of the code we are analysing. In effect the full description of the call might be:
	`--# derives Internal_State_Of_The_Run_Time_System from *, S &`
	`--#  S from ;`
	For our purposes it is enough to note that calling Set_True or Set_False unconditionally assigns a new value to the suspension object passed. It is better that the annotation accurately describes this behaviour than that it is consistent with the parameter mode.

R5	`Current_State` has been eliminated not just because it *can* create race conditions but because it *almost always* will. Essentially the places where it could be used safely (in protected code of a high enough priority to ensure that pre-emption does not occur) are so limited as to make it not worth retaining given the risks inherent in its use.
R6	This is for reasons of clarity and for consistency with sequential SPARK which, for example, requires pragma Import to follow the subprogram to which it relates.

Identifier	Rationale
R7	This is to ensure that atomicity is consistent and unvarying. If an atomic *object* is passed as a parameter then its atomicity can change. By ensuring that atomicity is bound to a *type*, we avoid these difficulties.
R8	`E'Count` can only return 0 or 1 in a Ravenscar program and so is not especially useful. Furthermore, it is difficult to model effectively for flow analysis purposes since the result must be obtained *from* something and that something is the internal state of the run-time system which we do not want to bring inside the SPARK boundary of the system.
R9	To allow a delay part way through evaluating an expression involving a function call would introduce evaluation order dependencies to SPARK.
R10	The transitive delay annotation allows the detection of blocking without requiring access to the entire program call tree. It is required, like other SPARK annotations, to allow analysis which requires only the specifications of referenced packages.
R11	In order to detect priority ceiling violations the Examiner must be able to identify which global variables are being referenced by a task object. The check is performed at the declaration of the task object. At this point its (task) type is known and since this must be declared in a package specification its type declaration will be available to the Examiner. The global annotation of the task type identifies (transitively) the state it references and/or updates and so it is possible to check the priority (if any) of this state for ceiling violations. The visible global annotation also allows a check, at the main program level, that tasks do not share unprotected state (see R18).
R12	Identifying, at the package specification level, what objects each task suspends on allows a check, at the main program level, that there is no possibility of more than one task suspending on the same suspension object or entry. Such multiple suspensions are prohibited by the Ravenscar profile. Without the suspends property, performing the check would require access to a complete linkable closure of the program.
R13	The use of an explicit (rather than default) priority, together with a fixed location for it, improves clarity and simplifies the construction of supporting tools such as timing and schedulability analysers.
R14	Simplifying the form of expression allowed in priority pragmas greatly simplifies the Examiner tool at no cost in power of expression. A pragma argument is a general Ada expression rather than a sequential SPARK expression. To evaluate such expressions would require adding a complete Ada static expression evaluator just for this pragma. If evaluated priority values are required they can be readily obtained by declaring a constant with the required evaluated value and using that constant as the argument to the priority pragma.
R15	A basic “sanity check” rule designed to avoid the gratuitous construction of useless program fragments.

Identifier	Rationale
R16	These rules are sufficient to prevent race conditions during elaboration. If a task imported unprotected state that is initialized during package elaboration then there is no guarantee (in the absence of the not-yet-standard pragma Partition_Elaboration_Policy) that the initialization will be complete before the task starts to run. A task may export an unprotected variable but, again to avoid race conditions between elaboration code and task execution, this should be the *only* way in which the variable is defined. The rule simplifies to "a task that uses unprotected state is solely responsible for its initialization". Similar issues occur for protected state. The intention is that protected state allows communication between tasks and so it is essential that it is initialized before concurrency occurs. The only way of ensuring that is to initialize it statically at the point of declaration.
R17	This is a natural extension of the general SPARK principle of avoiding anonymous types.
R18	Each task object is type announced with its type so that we can identify, without access to package bodies, each instance of each task type in the program. Together with the requirement that task types are declared in package specifications (see R11) this facilitates the main program check that task objects do not share unprotected state. The announcement of the type of task objects, together with the visible global and derives annotations of the task types, is also essential for the construction of partition-wide information flow relations for the program.
R19	Since the task type is declared in the package specification and the task body is in the package body, each will have different views of any abstract state in the package. The second annotation in this case is directly analogous to subprograms with the first annotation being expressed in terms of abstract own variables and the second in terms of their refinement constituents. Similar considerations govern protected bodies; the protected elements of the protected types are equivalent to refinement constituents of an abstract package own variable and a second annotation in terms of these constituents is required.
R20	Required to ensure that tasks do not terminate. Task termination would lead to implementation-defined behaviour.

Identifier	Rationale
R21	With the exception of partition flow analysis of interrupt handlers (see R27), this rule is perhaps not strictly required. However, it has some significant benefits and few drawbacks. Firstly, it is already a requirement for task types and would have to be a requirement for protected types that include interrupt handlers, so it seems easier to have the same rule for all the major new Ravenscar constructs. Secondly, and more significantly, SPARK simplifies the visibility rules of Ada in various ways that would be compromised by allowing protected type declarations and protected body declarations in the same package body unless additional rules were imposed. SPARK does not currently allow unit *specifications* to appear in package bodies with the sole exception of embedded packages. To allow embedded package specifications we regard package boundaries as being stronger in SPARK than they are in Ada. For example, an embedded package cannot see into its surroundings unless it inherits the enclosing package and uses qualified naming to reference entities in it. Allowing protected type declarations into package bodies would have required a similar strengthening of their boundaries and led to the need for inherit annotations on protected type declarations. All these complexities are avoided by the requirement to put the declarations in the package specification. Note that they may go in the package's private part and there is no requirement that protected *objects* be made visible.
R22	The Ravenscar execution model envisages the use of protected state to allow communication between tasks. If a protected object was allowed to interact with unprotected state it would become impossible, in general, to detect unsafe concurrent access to unprotected state. The RavenSPARK model is quite clear here and provides all the required functionality. (1) A task may make sole use of unprotected state. (2) Protected state can be safely shared. (3) A protected type may make use of unprotected state via the *protects* mechanism which guarantees that access to that unprotected state can only be made via the operations of a single protected object of that protected type. There is therefore no value in allowing unsafe access to unprotected state from within protected objects.
R23	This is a consequence of the general SPARK principle of avoiding overloading of subprogram names. If a protected type in package P exports an operation K and the package also exports a K then there are places in the package body where both operations will be visible and have the same name. The restriction is slightly unfortunate because a common paradigm is to provide operations in a package specification which are implemented by calling a suitable protected operation in the package body. Ideally we would like to use the same name for both but cannot because of this rule. Overall we judge that this minor inconvenience is outweighed by the benefits of avoiding overloading.
R24	The validity of calling interrupt handlers as normal subprograms is implementation-defined. It also has other problems such as being potentially blocking because an interrupt handler operates at a high priority level.
R25	A design goal of RavenSPARK is to be able to detect all violations of the rules of the Ravenscar profile statically. Clearly this is unachievable if, for example, priorities are set dynamically.

Identifier	Rationale
R26	We need to know from package specifications whether an own variable is potentially shareable. The *protected* modifier provides that information.
R27	This rule is needed to support partition flow analysis and is analogous to R18 requiring type announcement for own task variables. Interrupt handlers are somewhat like tasks in the sense that they may execute autonomously and periodically. From the main program, and with access to package specifications only, we can identify all the protected objects in the partition which contain interrupt handlers (via the interrupt property). The type announcement and the requirement that protected type declarations are declared in package specifications gives us access to the global and derives annotations associated with each potential interrupt. This information is required to construct the partition flow relation. Strictly we only need the type announcement if the interrupt property is present but it is simpler to require it for all objects of a protected type.
R28	For cross checking against a task's *suspends* property annotation
R29	Practically this is to facilitate the construction of shareable, protected ports. From a language design point of view it is directly analogous to sequential SPARK's refinement rule that allows an abstract own variable to have a mode if all its refinement constituents also have the same mode. Here we are saying that protected object may have a mode if all its protected elements (which are its refinement constituents) have the same mode. Note that it follows that all the protected elements must be *virtual* protected elements since this is the only way a protected element can be given a mode.
R30	Virtual protected elements are a mechanism to allow a protected object to control access to a variable that ideally we would have made a protected element of that object but cannot for some reason, perhaps because it requires an address clause. Since we are binding the protected object and the variables(s) it is protecting tightly together it is appropriate that they should be declared in the same package; this also makes it possible to check that virtual elements are not being misused without requiring access to an entire linkable closure of the program. We require virtual protected elements to be initialized at declaration (not in package elaboration code) because they are effectively protected state and all protected state requires static initialization to avoid race conditions. Moded virtual protected elements are deemed to be initialized by the external environment (in common with external variables in sequential SPARK). Only one instance of a protected object of a protected type which includes the protects property is allowed because otherwise the two objects would be sharing unprotected state (while at the same time claiming sole ownership of it). Since a virtual protected element behaves in all respects in the same way as a protected element it follows that it is visible only in the protected body of the type that protects it. This rule is required to avoid unintentional sharing of unprotected state.

Identifier	Rationale
R31	Allowing non-scalar types to be qualified with pragma Atomic would introduce implementation dependencies. It also undermines atomicity if we, say, make record R with more than 1 field atomic and allow one task to access field R.F while another attempts to write to R.G. RavenSPARK envisages the use of pragma Atomic variables for simple items such as flags and counters; more complex data should be handled by a user-defined protected type.
R32	These rules are identical to sequential SPARK's rules for external variables and are intended to avoid evaluation order dependencies and to avoid information flows being generated by *expressions* (rather than by *statements*). A simple example, if X is a shared variable of a pragma Atomic type then the expression "X < X" can return `True` or `False` depending on the order of evaluation and whether the value of X is changed by some other program thread during the evaluation of the expression.
R33	Within the protected sequence of statements the variables cannot be accessed by any other program thread so they behave just like a variable in sequential SPARK and require no special rules.
R34	This is required to prevent program termination. If the program includes tasks then these will have a plain loop which prevent them terminating (see R20). In this case, the main program (more correctly, the environment task) will execute and then wait on its end statement for the tasks to terminate (which they won't). If, however, the program contains no tasks then the main program would execute and, on reaching its end statement, halt the entire program's execution. We therefore require at least one plain loop in the program, either in a task or in the main program itself.
R35	The Ravenscar profile execution model is built round a fixed set of tasks co-ordinated via a fixed set of protected objects. Allowing the aggregation of task and protected objects into larger data structures or their passing as parameters runs counter to this simple model and introduces unnecessary complexity to the analyses that can be performed. For example, the information flow relation for a procedure which exports an array of protected objects each element of which contains a number of entries and/or interrupt handlers is rather obscure!
R36	Protected types provide a template for a set of operations on data but the actual data resides in *objects* of the protected type. To perform flow analysis of protected operations, we need a way of expressing global and derives annotations that is independent of the actual objects that will eventually be declared. The use of the type name for this purpose was chosen in preference to adding a new reserved identifier such as "this" to the SPARK language.

Identifier	Rationale
R37	SEPR 2253 adds the case of an Atomic record type if the record is not tagged and has a single field ~~which~~ that is a predefined type. Without this addition, it was impossible to have an Atomic object of a predefined type such as Boolean, Character, or Integer. ~~-~~The types permitted inside type a record include all the scalar types declared in Standard (including Duration and any additional Long_ and Short_ Integer or Float types introduced in the configuration file), plus ~~System.Address,~~ Ada.Real_Time.Seconds_Count, ~~and~~ Ada.Interrupts.Interrupt_ID, and (as a special case) System.Address~~.~~

Document Control and References

Praxis High Integrity Systems Limited, 20 Manvers Street, Bath BA1 1PX, UK.
Copyright © Praxis High Integrity Systems Limited 2008. All rights reserved.

Changes history

Issue 0.1 (26th February 2003): First draft

Issue 0.2 (7th March 2003): After initial comments from Keith Banks and Brian Dobbing.

Issue 0.3 (25th March 2003): After first formal review

Issue 0.4 (2nd April 2003): Cross-checking of grammar with SPARK report and minor changes.

Issue 0.5 (14th May 2003): Incorporation of comments by Jonathan Hammond

Issue 0.6 (15th May 2003): Addition of design rationale section.

Issue 1.0 (15th May 2003): After review of 0.6 by Jonathan Hammond.

Issue 1.1 (23rd November 2004): Company name change only.

Issue 1.2 (22nd December 2004): Definitive issue following review S.P0468.79.88.

Issue 1.3 (22nd November 2005) Line Manager change.

Issue 1.4 (29th November 2005) Updated following review S.P0468.79.90.

Issue 1.5 (7th December 2006): Update to definition of Ada.Real_Time package (SEPR 2081).

Issue 1.6 (25th January 2008): Update to allow single-field records to be Atomic (SEPR 2253).

Changes forecast

To be kept in line with future language and Examiner developments.

Document references

1 Guide for the use of the Ada Ravenscar Profile in high integrity systems. A.Burns, B. Dobbing and T. Vardanega. University of York report YCS 348 (2003). Available from: http://www.cs.york.ac.uk/ftpdir/reports/YCS-2003-348.pdf

2 SPARK 95 the SPARK Ada95 (with Ravenscar) Kernel

3 Real-Time Systems and Programming Languages, Alan Burns and Andy Wellings, Third Edition. Addison Wesley. ISBN 0 201 729881 1.

4 The INFORMED Design Method for SPARK, Peter Amey, Praxis High Integrity Systems

5 RavenSPARK Design for the Mine Pump Case Study, Keith Banks, Praxis High Integrity Systems